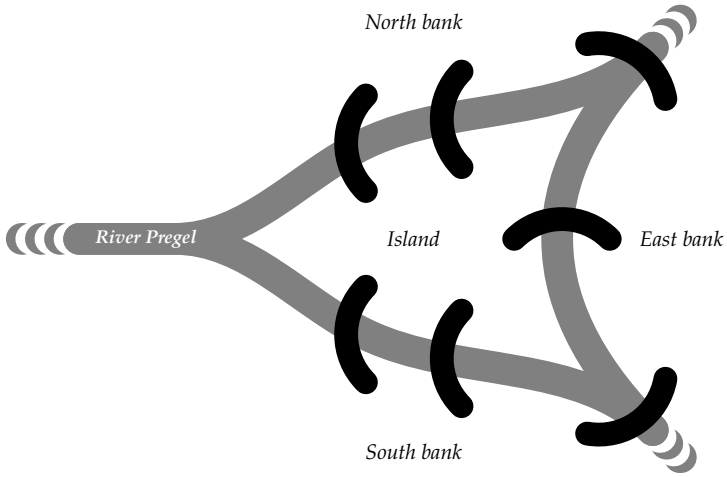
8 Graphs

All sorts of things can be represented by dots and arrows, from the world wide web down to how the neurons in our brains are connected. Interactive devices are finite state machines that can be represented by dots and arrows. Common to all is graph theory, the underlying mathematical theory. Graph theory gives us a powerful handle on what devices are doing, how easy users will find things—and it gives us lots of ideas for improving designs.

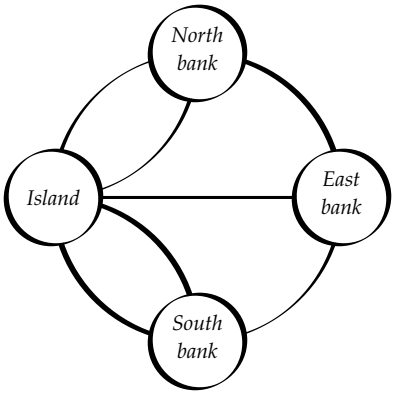
In the eighteenth century folk in the German town of Königsberg entertained themselves walking around the town and crossing the bridges over the River Pregel, which empties into the Baltic Sea. There were seven bridges over the Pregel, an island, and a fork in the river. Some wondered whether it was possible to walk around the city center crossing each bridge exactly once.

Here is a schematic representation of the city of Königsberg, its river and its bridges schematically:



You can imagine that somebody walking around Königsberg is in different states depending on which bit of dry land they are on—north bank, island, south bank, and so on. Crossing a bridge changes state.

We can redraw Königsberg without the river and draw circles around the states, and we get something closer to a familiar state transition diagram:



One frustrated person who tried walking across each Königsberg bridge exactly once, perhaps on some romantic evening stroll, wrote to the local mathematician Leonhard Euler about the problem. By 1736 Euler had published a paper that showed conclusively that there was no way to walk around the city crossing each bridge exactly once.

Euler's 1736 paper started the field of graph theory, which thinks abstractly about networks, like walks and bridges. In Euler's honour, a walk that crosses each bridge exactly once is called a Euler Tour, and any town with bridges that can be walked around in this way would be called Eulerian. Today, the Königsberg bridge problem makes a nice children's game—there are a few mathematical addicts who have built small-scale models of Königsberg for children to enjoy running around them—and maybe learning a bit of mathematics as well. Another bridge was built in Königsberg in 1875 and it then became possible to walk around the city center crossing each bridge exactly once: so post-1875 Königsberg is Eulerian.

The point for us is that graph theory analyses a network, here the bridges of Königsberg, and tells us what humans can do with it. In this chapter we will explore graph theory to see what it can say about interaction programming, and how design choices influence what a user can do and how easily they can do things.

■ ■ ■

Euler's name is pronounced "oiler," but Eulerian is pronounced "you-lair-ian." In graph theory "bridge" is a technical term that means something very different from a river bridge. To keep the confusion going, after World War II, Königsberg was renamed Kaliningrad.

8.1 Graphs and interaction

People in Königsberg had set themselves a certain sort of navigation task: to walk around a city under certain constraints—and some people got frustrated when it turned out to be too difficult to do. Today users navigate around interactive devices, pressing buttons and changing states, rather than crossing bridges and changing river banks. They set themselves tasks, to achieve things using the devices, or they just want to enjoy themselves as the people of Königsberg did. The underlying mathematical theory is the same.

A graph is a set of things, conventionally called vertices and a set of pairs of those things, usually drawn as lines between them, called edges. These are just alternative words for what we called states and actions in chapter 6 (p. 163). As far as graphs are concerned, states are vertices, and actions are edges.

In this book we are concerned with one-way actions, which are drawn as single-headed arrows: in this case the graphs are called directed graphs, and the directed edges are usually called arcs. Almost every diagram we've drawn in this book (apart from the two opening this chapter) is a directed graph.

Modern Königsberg has some one-way streets, and this makes it a mixed graph, as it has both directed and undirected transitions; however, none of the systems we look at in this book are mixed graphs.

Graphs get everywhere, and this chapter is where we pin down what can be done with them. If you thought that something as simple as graphs—dots and arrows—could not be useful for interaction programming, you'd be wrong! Even very simple graph properties have serious usability implications.

Graphs can describe many different things. They have many interesting properties and anything described by a graph, whether a garden maze or an interactive device, can be reasoned about as a graph and will enjoy the same sorts of general properties. It's rather like a pile of oranges and a pile of apples both enjoy the basic properties of numbers (if you add a piece of fruit, the pile goes from n to $n + 1$ pieces; of if there are n pieces of fruit, you can't take away more than n pieces; and so on)—anything that is a graph, or that can be understood as a graph, enjoys graph properties.

Humans made a giant leap forward when they realized that two apples and two hungry people both had the same property—two. When you know that you can use mathematical properties of numbers, for instance, to deduce that each of two people person can have one apple (from two) to stave off their hunger. Similarly, once we realize that lots of human experience is basically like finding your way around in graphs, we can apply more powerful reasoning—in the same way that numbers, once we get to grips with them, let us add 1 to 999,983,142 as easily as to 2.

Let's begin with an example from a completely different area, well away from interactive systems—chemistry. A compound like paraffin can be represented as a graph, where vertices are atoms and edges are chemical bonds between the atoms.

One of the questions chemists are interested in is how many isomers a compound has: that is, how can the same numbers of hydrogen and carbon atoms be

combined in different ways, perhaps each with different chemical characteristics? The graph theory here gets going when it starts counting off how many chemicals have particular properties; historically, graph theory got really going when it brought some clarity to chemical thinking.

Euler's original bridge-crossing problem corresponds to a modern problem, and one of concern for us. Suppose you give me a gadget and ask me to check to see whether it works properly—that it works correctly according to its user manual or specification. (Or you could give me a web site and ask me to check it; since both things are directed graphs, the problems are equivalent: both devices and web sites are graphs.) So I am supposed to press buttons and note what they do, and I must check that in every state every button behaves properly.

Of course, when I press a button on a gadget, my action changes the device's state. It's analogous to crossing a bridge in Königsberg—but instead of walking around the city, I am walking around the space of device states; instead of changing my location to different places in the city, I am pressing buttons that cause the state of the device to change. The only real difference is that in Königsberg, if I make a mistake I can turn round and go back over the bridge, but when testing a gadget there may be no way back—it's as if the bridges on the device are all one-way. In short, to test a device against a user manual efficiently, without recrossing "bridges," we would like to find an Euler tour of the device.

Unfortunately not all graphs have Euler tours. So we are being unreasonable to hope to be shown an Euler tour to follow to test the device.

8.1.1 The Chinese postman tour

The next best thing to an Euler tour is a Chinese postman tour, so called because it's named after its Chinese discoverer (Kwan or Quan), and it's called the postman tour because postmen walk down every street in a city to deliver letters. The postman wants to walk the shortest distance, not going down any streets more than strictly necessary. This simple generalization of the Euler tour was only discussed in the 1960s.

The Chinese postman tour turns out to be quite tricky to find by hand, but a computer can find the best Chinese postman tour of a graph easily. Once we have a Chinese postman tour, when you ask me to check a gadget I could follow the postman tour, though I might need a computer to help me work it out.

Left to my own devices, I would risk going around and around in the state space and most likely never managing to check all of the device thoroughly. If you want to check that a device does what you as its designer intended it to do, you should not just ask users to try it out; rather, you should work out a Chinese postman tour and check that—that will be a lot quicker and far more reliable. Of course the Chinese postman tour check would not confirm that users can do all the useful tasks they want to do, but it will make a basic (and necessary) check that the system is correctly designed. For example, a Chinese postman tour would check on a web site that every link was correctly named and went to the correct page.

Box 8.1 Chinese postman tools If we ask users to test a device, effectively to check "does this bit do what its manual says?" and "can you think of any improvements?" for every function available—without further help—they are *very* unlikely to follow an optimal way of testing the device. They are very unlikely to check every action (every button press in every state): they will miss some. In the very unlikely event that they do manage to check everything, they certainly won't do it very efficiently. They will use up a lot more time and effort than necessary. Trial users checking systems often have to use notebooks to keep track of which parts they have tested, and I doubt they can be sufficiently systematic even then not to waste time and miss testing some features.

In short, a program should be used to help users work through their evaluation work. This means finding a Chinese postman tour. Actually, rather than follow an accurate Chinese postman tour, it's more practical to follow a dynamically generated tour that suggests the best actions to get to the next unchecked part of the device. This is much easier! For the program, it only needs to keep track of which states have been visited and use shortest paths to find efficient sequences of actions for the users. For the users, it's much easier too: there is no problem if they make a mistake following the instructions, and they can easily spread the testing over as many days or weeks as necessary.

▷ Section 9.6 (p. 297) gives code to find shortest paths. The further reading (page 271) gives a reference with Java code for the Chinese postman tour.

Since a device design may change as it is developed, the flags associated with states can be reset when the design changes if the change affects those states. Thus a designer can incrementally check a device, even while it changes, perhaps making errors or missing actions, and still know what needs evaluating and checking. Eventually the evaluation will cover the entire functionality of the device.

State flags can be used in two further ways. During design, documents may be produced, such as user manuals. A technical author may wish to flag that they have documented certain parts of the device and therefore that they want to be notified if those parts of the device ever change. This allows a technical author to start writing a user manual very early in the design process. State flags can also be used by an auditor, who checks whether an implementation conforms to its specification. The auditor can use the flags to assert that a vertex (or arc) has been checked out and must not be changed gratuitously.

The Nokia 5110 phone that was used in chapter 5, "Communication," has a Chinese postman tour of 3,914 button presses! And that's only considering the function menu hierarchy, not the whole device. In other words: to test whether every button works in every state on this device takes a minimum of 3,914 presses—and that's assuming error-free performance. If we made a slip, which we surely would, it would take longer. Obviously, human testing is not feasible.

Here is an extract from a Chinese postman tour of a medical device:

```
    :
478 Check ON from "Off" goes to "On"
479 Check DOWN from "On" goes to "Value locked for On"
    In state "Value locked for On", check unused these buttons do nothing:
    DOWN, OFF, PURGE, UP, STOP, KEY, ON
487 Check ENTER from "Value locked for On" goes to "Continuous"
    :
```

You can see how each check takes the user to a new state, and in that state only one check can be made there—you need a Chinese postman tour because each check necessarily takes you to a new state. In the example above, step 479 must be the first time the state `Value locked for On` has been reached in the tour, and the checking program is recommending that several buttons are checked in this state to confirm that they do nothing: these buttons can be checked easily since checking them shouldn't change the state.

▷ With the device framework we will develop in chapter 9, "A framework for design," it is easy to take the test user "by hand" through the best tour: the device itself can keep track of what the user has and has not tested. This idea is explained more fully in box 8.1, "Chinese postman tools" (p. 229).

If the system can keep track of all this, why does a user have to work laboriously through the tour? The system can check that everything is connected, but no system can know deeper facts like what the Pause button is supposed to do when the device is recording. A user should get the device in the recording state and try out the Pause button, see what it does, then tick a box (or whatever) to "audit" the correct behavior of the device. Once a user has checked the Chinese postman tour (or any other properties), if the device is modified only the bits that have changed need to be rechecked—again, a good device framework can handle this and greatly help in the design and evaluation of a device.

The longer the Chinese postman tour, the harder a device is to check properly: it simply takes more work. A designer might therefore be interested not in the details of the Chinese postman tour (because the details are only needed for checking) but in how long it is, that is, how many button presses it will take to do, since that is a good measure of how hard the device is to understand. If a user was going to understand a device they would *have* to explore and learn at least as much of the device as a postman would have to, so the length of the postman tour provides a way to assess how hard a device is to understand or to learn. Even if this is not exactly correct (users may not have to know *everything*), the longer the tour the harder the device (or the web site, or whatever) will be to check. So, when a designer is contemplating adding a new feature or a new button, they might want to measure the lengths of the Chinese postman tours with and without the feature to see how the lengths change. Is the extra cost to the user in terms of complexity worth it in terms of the value of the new feature?

Some special graphs have simple Chinese postman tours: a so-called randomly Eulerian graph, for example, has a tour that can be found by pressing any button that has not yet been pressed in the current state. It's called randomly Eulerian because when you have a choice, you can behave randomly and still follow a Eulerian tour of the system. Such devices are very simple. If you had a design like this, its usability could be improved (if seeing all of it is the task—as it would be with a task like visiting an art gallery and wanting to walk down every passage to see everything) by having the system indicate which buttons have previously been pressed.

Art galleries might like to ask design questions like these. Presumably they want visitors to see many exhibits, even—and perhaps especially—when the vis-

itors get lost. Equally, the designer of a walk-up-and-use exhibit at an exhibition might want to design their system's graph to be randomly Eulerian, guaranteeing that the user will be able to find more things on the console more easily.

8.1.2 The traveling salesman problem

More familiar to most people than the Chinese postman tour is the traveling salesman problem. We imagine that a salesman wants to travel to every city to sell their wares. Now the cities in the salesman problem are the vertices of the graph, and the roads between the cities are the edges (or if they are one-way roads, they are the arcs).

Unlike a postman, a salesman is not worried about traveling along every *road*—just about visiting every *city*. But like the postman, the salesman wants to travel the shortest possible distance.

If we think about web sites, the traveling salesman problem corresponds to checking each *page* of a web site rather than each *link*, which is what the postman tour checks. The traveling salesman tour of a general interactive device can check that each state of a gadget works as intended—for instance, that the various lights and displays come on properly in every state. In contrast, the Chinese postman tour checks whether the buttons (or other user actions) work correctly. Thus to check that a device works correctly, we need to check both its Chinese postman tour and its salesman tour.

As it happens, a Chinese postman tour must visit every vertex, so a user (or a designer) can check out every state by following the route of a postman: you don't need to do both. If we have a test rig for the device being checked, then it should be extended so that when a user visits a state for the first time, they are asked to check whether it is doing what it is supposed to do. When the test user checks the box, they need not be asked again even if they revisit the same state. Similarly, the first time they press a button in each state, they should be asked, but not subsequently.

The traveling salesman tour problem is not at all easy to solve, and computers get overwhelmed with relatively few vertices—the hundreds or more states of a typical interactive device would be way beyond finding an exact solution. In practice, the salesman therefore has to make do with an approximate solution, hopefully one that is good enough. But just like the Chinese postman case, if we are checking a gadget or a web site, *we* would take a very long time if we got lost without following the route instructions worked out by a program.

Compared with the traveling salesman, who visits everywhere, how well does someone using a device typically do? Or, put another way, if we ask users to try out a gadget so we can see how they do, so we can evaluate the device design, what proportion of the system will they explore? If users testing a system don't know where to go (if they are not following a recipe, such as a salesman tour), just as lost people in the real world, they will probably go around in circles and never get to explore the gadget at all thoroughly. This is a good reason not to rely exclusively on user tests for evaluating a device.

Euler tour To travel along each arc (in the right direction) exactly once, and get back to the starting vertex.	The graph must be Eulerian, and if so, checks every action is correct.
Chinese postman tour To perform a Euler Tour, or when an Euler tour is not possible, to travel along each arc at least once.	To check that every action (button press; link) is correct.
Hamiltonian tour To visit every vertex exactly once, and get back to the start.	The graph is Hamiltonian.
Traveling salesman tour To visit every vertex, using as few arcs as possible.	To check that every state (or web page) is correct.

Figure 8.1: Summary of four sorts of graph tours. The Chinese postman and traveling salesman tour have variants if arcs are weighted (have numbers associated with them): then they should minimize not the number of arcs used, but the total cost of the weights.

- ▷ The cost-of-knowledge graph plots how long a user takes against how much of a device they have explored. If the user follows a traveling salesman tour, they will do the best possible. Section 5.3 (p. 144), specifically, figure 5.11 (p. 144), draws some graphs, from random user behavior.

8.1.3 Coloring graphs

A long time ago, map makers noticed that only four colors are required to make a map with no adjacent countries or regions the same color. The only map we've drawn in this chapter, of the town of Königsberg, with four regions of land, would only take three colors: if somebody had painted the town, three colors are enough to ensure that the land color changes whenever you cross a bridge. The north bank has no bridge connecting it to the south bank, so these can be the same color. For more complex maps, you may need more colors. The four color theorem says, against expectations, that you never need more than four, provided the map is flat.

The four color property of maps can be expressed in graph theory: regions are vertices, and "being next to another region" is an edge between the regions or vertices. Although first proposed in 1852, it was over a century before it was proved. The proof of what seems like a trivial theorem turned out to be so complicated that it had to be done by computer—indeed, the computerized proof was quite controversial and called into question what mathematicians mean by proof: are proofs supposed to be clear arguments understood by humans, or can computers get away with vast so-called proofs that no human could verify?

Now exactly what has coloring got to do with interaction programming? Imagine, first, that you are back in Königsberg. (It's tempting to call the regions in this land "states" because that's what we will be calling them in a minute!) If the land had been colored by a painter who had arranged that no adjacent lands were the same color, you would always know when you crossed a bridge. If the painter hadn't used enough colors, sometimes you would cross a bridge and the country

color would not change. You might not notice that you were in a different country, and perhaps wouldn't understand the local culture's colorful nature.

Back in the context of user interfaces, users would probably want to see confirmation every time they changed the state of the device. In some sense, the "color" of each state must be different from the "color" of adjacent states; if not, then some state changes cannot be noticed, because there is no change.

What corresponds to colors in an interactive device? Most devices have displays and indicator lights that help tell the user what state the device is in. For example, there may be a light that says the device is on. This is rather like saying that in the off state, the light is black, and in the on state (whichever state it is in when it is on) the light is red. A CD player might have an indicator to say whether a CD is in the device or not: another choice of two colors. The combinations of indicators correspond to colors: in the table below, each combination of the two indicators has been allocated a notional color.

State	*Indicators*	*Color*
Off	none	Red
On, CD out	on	Green
On, CD in	on, CD	Blue
Off, CD out	CD	Yellow

Every possible state change (like ejecting the CD) corresponds to a color change. In fact, for this simple abstract device, since every state is a different color, it's inevitable that any state changes the user can do results in color changes. In general, what a designer should check is that at least one indicator changes for all possible state changes; then the device is adequately colored, and in principle the user can tell whenever there is a state change.

▷ Section 10.3.2 (p. 334) shows how to work out tables like the one above, automatically so that they can be used to help design better systems.

The four color theorem only applies to flat, conventional, maps. For more complex graphs—such as those we encounter with interactive devices—almost any number of colors may be required. The worst case is a complete graph: a complete graph has every vertex connected to every other, so that all vertices must be different colors. If there are N states, N colors are required.

A graph's chromatic number is the minimum number of colors it needs. The chromatic number of any two-dimensional map is at most 4, unless it has tunnels or other ways of cheating. The chromatic number of any map drawn on a sphere is 4 too but the chromatic number of a map drawn on a torus (a shape like a sausage bent into a circle) is 7. An interactive device whose graph can be drawn on paper without any crossing lines will necessarily have a chromatic number at most 4, but usually the chromatic number will be higher—most device's graphs cannot be drawn without crossing lines.

For any specific graph, it is possible to work out the chromatic number—the programming is not too hard. If the chromatic number is c, then there must be enough indicators to count from 1 to c (or, equivalently, from 0 to $c - 1$) in binary. Imagine each state is given a color. The chromatic number tells us the least number of colors we need so that when we go from one state to another, the color changes.

If our device had a single colored light with c colors (or c different phrases), that would be sufficient to tell the user whenever they changed state. With fewer than c colors, some state transitions could be missed using the indicators alone: the light's color wouldn't change.

Now it is unusual to have a single indicator with lots of colors; instead, there are usually several indicators, red lights or whatever, that are either on or off, rather than showing different colors. In principle, the indicators can count as binary digits, bits. If you have n indicators, you effectively have n bits available, and you can count up to 2^n different things. This is how binary works. So you need 2^n to be at least as large as c to be able to distinguish all state changes. In other words, you need at least $\log_2 c$ on/off indicators to distinguish every state for a device with chromatic number c. Even then, the indicators may not be used properly (the device might have the right number of indicators but not use some of them or not use them very well), so that needs checking as well. That is, having enough indicators is necessary to distinguish all the states, but it is not a sufficient design criterion to ensure that the device is easy to use.

For a JVC video recorder I worked out the chromatic number, and it was higher than the number of indicators could handle. In fact, the video recorder did not say when it was in the states "preview" and "cue." That's a potential problem identified by graph theory—though whether a user would be worried about it is another matter. For a video recorder, a user would probably be looking at the TV, not the device itself, and it would be obvious whether it was playing or fast forwarding from the TV picture. Once or twice, though, I have found myself fast forwarding the tape when I wanted to rewind it and not noticing the difference.

▷ This JVC PVR is discussed in detail in section 10.3 (p. 330).

For a safety-critical device (say, a medical device to be used in an operating theater), knowing when or whether it changes state is likely to be very important. What happens if an anesthetist presses 8.0 to deliver 8.0ml of a drug but accidentally does not press the decimal point hard enough? The patient would get 80ml of the drug, which for many drugs would be quite enough to kill the patient. In such cases, the device should tell the user, the anesthetist, *every* time it changes state—so, in particular, not pressing the decimal point hard enough should be recognized by the user.

8.1.4 Coloring graphs to highlight interesting features

Any graph I draw in this book is going to be small, but I hope you agree, worth studying. If it's too big to draw, I won't include it in *Press On*; and if it's a boring graph, there's no point in drawing it.

Unfortunately, the real world has a lot of enormous and potentially boring graphs in it (the web may be a good example). Fortunately, we can use coloring as a way to transform the boring world into exciting and manageable small graphs.

Imagine a huge graph, for instance, one of the enormous state machines from the previous chapters. We color everything we are really interested in red, and

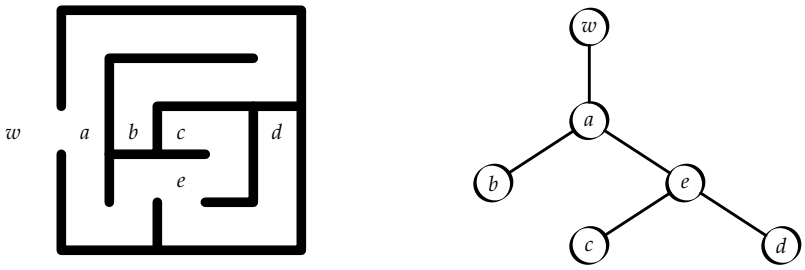


Figure 8.2: A very simple maze (left), with intersections and end points marked, and its corresponding graph (right). The state *w* represents being in the "rest of the world," and the states *a* to *e* are places to be inside the maze, where the explorer has to make a choice or has come to a dead end. The graph representation, being more abstract than the maze, does not distinguish between the inside and the outside of the maze—for instance, it could be redrawn with *c* or *b* at the top of the diagram.

leave everything else black. We now have a colored graph, that is a lot simpler than the original. Hopefully, we will have colored in something that has an interesting structure that we can critique, understand, and improve.

For example, later in this chapter we will talk a lot about a type of graph called a tree. Many interactive systems considered as state machines are *almost* trees, but they have details that make them more complex. So imagine that the tree part of the graph is colored red. What we want for good interaction programming is a *good* red tree. The other stuff, which we haven't colored red, can wait, or perhaps it can be handled automatically and generated by a computer—for instance, features for error correction. Then we look at the red tree, ignoring all the other detail, and try to improve the design of the red tree.

▷ Trees are discussed in section 8.7 (p. 249).

8.2 Mazes and getting lost in graphs

Graphs get much more interesting when they are too big to look at. Whether using devices or testing them, users can get hopelessly lost, even in quite small graphs, even comprising only a dozen states. You can write programs that manipulate graphs of millions of states, and typical interactive devices have thousands of states. Is there anything else from graph theory that can help?

When the Attic hero Theseus explored the Minotaur's labyrinth, he unwound a cotton thread so that when he wanted to get out, he could be guided out by following the thread back to the entrance. If he hadn't had the thread, he would have become lost—and he would have figured on the Minotaur's next menu.

When our contemporary hero explores an interactive device, say a ticket machine, what "threads" can prevent getting lost? What can they do if they take a

wrong turning? What happens if they fall through a trap door that only lets them go one way and not back?

The answer is that an interactive device should provide something analogous to Theseus's thread. Following the thread back will always get the user out. It's easy enough to do this. The thread is usually called "undo." When users get lost, they press [Undo] and get pulled back by the thread to where they just were.

There are a few complications, most notably when the user undoes back to a state where they have been twice (or more times) before. It isn't then obvious what pressing [Undo] should do—it's a state where the thread crosses over—go to the previous state, or to go to the state that first got there. There's a sense in which, if the user is trying to recover from a mistake, simplifying the undo trail will help; on the other hand, any simplification is second-guessing the user and may contribute to further problems.

When Theseus got back to any doorway he'd been through twice before, he would see threads going in three directions, and he'd run a small risk of going around in a circle and taking longer to escape the maze. Even if he had been using Ariadne's special directional thread (so he knew which direction he was going in when he unwound it), he would have two choices—or more choices in a popular intersection. The moral for Theseus is that whenever he *first* gets to an intersection, he ought to leave a stone behind to mark the first entrance, where he entered, to the intersection—and then it will be easy to decide which way to get out when the Minotaur chases him and he hasn't time to think hard about it.

The great thing about graph theory is that we know these two problems—Theseus panicking about the Minotaur and a modern ticket machine user (with a train coming to pick them up *if* they have managed to get a ticket out of the machine)—even though one is the stuff of myths and the other the stuff of modern stories and HCI textbooks—are precisely analogous. Thus a good ticket machine design will give the user different choices in different states, depending on how to get out. A ticket machine has to make other decisions: as the user travels around its "maze" they will be picking stuff up at different "intersections," like going to states where they can tell the ticket machine where they are going, how many tickets they want, how many children are traveling with them, and so on. When a user wants to "get out" not all of these details should be lost. Escaping from the labyrinth is easier in comparison: Theseus merely wants to get out fast, not get out with a valid ticket!

Another solution is to have more than one action available to the user. A key [Previous screen] is potentially ambiguous, since there may be two or more previous screens for the user—like having two or more threads coming to an intersection. We could design the device to have [Previous screen] and [Original way here], which (easily if it was a soft key) would only appear when there was a difference. Really the only way to tell what approach would help is to do some experiments with real people, under the sorts of pressure they'll be under when they need to buy tickets before the train comes.

It is possible to escape from certain sorts of real mazes without needing a thread or without needing an exceedingly good memory for places. In some types of

maze you can put your hand on the right hand wall or hedge, and keep on walking with your hand following the right hand wall. This will take you in and out of cul-de-sacs (dead ends), but eventually you will escape. If you happen to start trying to get out when you are in an island within the maze, all you would achieve then is to go around the island endlessly, never escaping. Hopefully, in real life, you'd notice this repetition and at some point change hands to touch a wall you'd never touched before. This would get you off that island, and perhaps closer to the exit.

In 1895 Gaston Tarry worked out a general technique for escaping from mazes. His assumption was that the passages in the maze were undirected, so you could walk along them in either direction. Of course, the actions in a state machine device are directed so Tarry's method does not work for a gadget, unless we design it so that actions are, like passages, undirected. This would mean that there needs to be an inverse for every action: if you can get to a state, you can always go back: to use Tarry's ideas, the device would need an Undo button.

Tarry had two ideas about goals. The simpler is that we want to find the center of the maze, and his systematic way to do this is to obey the following rule:

- Do not return along a passage that you took to an intersection for the first time unless you have already returned along all the other paths.

This would be one way for a traveling salesman to work if they didn't have a map of the country (or a map of the maze): eventually they will get to every city. This method of Tarry's eventually visits everywhere, so it will certainly get to the center of the maze. In fact, for a device, it is a technique for a user to get the device to do (or be able to do) anything it can do—provided the device has a proper Undo button.

His more complex idea was that from the center you would want to get out efficiently. The first method, above, of finding the center takes you all over the place; because you don't know where the center is you need to try going everywhere. But to escape faster, rather than reexplore everywhere, carry a bag of stones and mark each passage you travel down—roughly speaking you use the stones as marking the "ends" of a thread. If you are Theseus, you need a big reel of thread, at least as long as the passages put end to end; if you are Tarry, you need a bag of stones, big enough to leave stones at each junction you visit. Tarry's stones are equivalent to Theseus's thread, except no trail is left down the passages only at their ends. If you are a modern interactive device user, you just need the device to be designed to keep track of the paths that would have been marked with threads and stones.

I haven't given quite enough details, but Tarry's approach is equivalent to the following more abstract description. He effectively replaces each edge in the original maze with a pair of directed, one-way edges going in each direction. He uses the stones to keep track of which way you've been down any edge. His procedure then efficiently finds an Eulerian tour around this directed graph (there always is one as he's added edges)—you walk down each maze passage twice, once in each direction—so effectively along each virtual directed edge once.

What use are these graph theory ideas for interactive systems? It's interesting that web browsers typically change the color of links (hot text) when they have been used. This is like putting stones down in passage entrances: the links are

the ends of passages (to further pages on the web site) and the change in color marks the fact that the passage (or link) has been used before. Tarry himself suggested this is useful idea, and no doubt usefulness is what prompted the feature in browsers, though I don't think the designers of web browsers got the idea from systematically thinking through graph theory ideas.

Why don't interactive devices have lights on their buttons? Maybe the color of a button should change if it has been used before in this state? Maybe a button should flicker tantalizingly if the user has never tried pressing it in this state? Such a device would encourage and enable the user to explore it fully over a period of time. In fact, this idea is a bit like putting the user testing features of a framework (like the one we mentioned above for checking that the test user is following a Chinese postman tour) into the device itself. What's a good idea for the tester is probably good idea for the general user too.

It is important that there is an algorithm, Tarry's procedure, for getting around a maze. The algorithm does not rely on there being a map or bird's eye view or indeed having any prior knowledge about where the center of the maze is. This is rather like wanting to build a device that allows the user to succeed, but (even *with* a map of the gadget) not knowing what the user's tasks are. What state or states would the user consider the center of their maze of tasks? We don't know. What we do know, though, is that it is possible to help users to systematically explore a gadget so that eventually they can find anything they want in it (*provided* they can recognize it when they get there).

In complete contrast, recreational mazes are deliberately designed to be a challenge. Mostly when we design user interfaces we want to help users rather than challenge them, which suggests that rather than helping users cope with the current design, we could try redesigning to make finding our way around easier. Graph theory gives us several ideas to do so, from changing the structure of a device (so a particular way of navigating it will work), changing the actions or buttons available (so Undo works usefully), changing the indicators or lights at certain states (so users know they have been there), and so on.

8.3 Subgraphs

Most devices are more complicated than the conceptually simple sorts of graphs—complete graphs, cycles, or trees—yet these are clearly relevant and useful design concepts. How can we get the best of both worlds?

Inside a mobile phone, one would expect to find a cycle of menu items that the user can choose and a Scroll button or a pair of buttons ∧ and ∨ that cycle through them. Yet the device's graph as a whole is not a cycle: there is more to it.

A subgraph is part of a graph, obtained by taking some (possibly all) of the vertices from the original graph as well as some of (possibly all of) the arcs that connect them. Thus we expect a mobile phone, for example, to include a subgraph that *is* a cycle, namely, the cycle of menu items.

Subgraphs have other applications. From a usability point of view, if a user does not know everything about a device, but what is known is correct, then they know

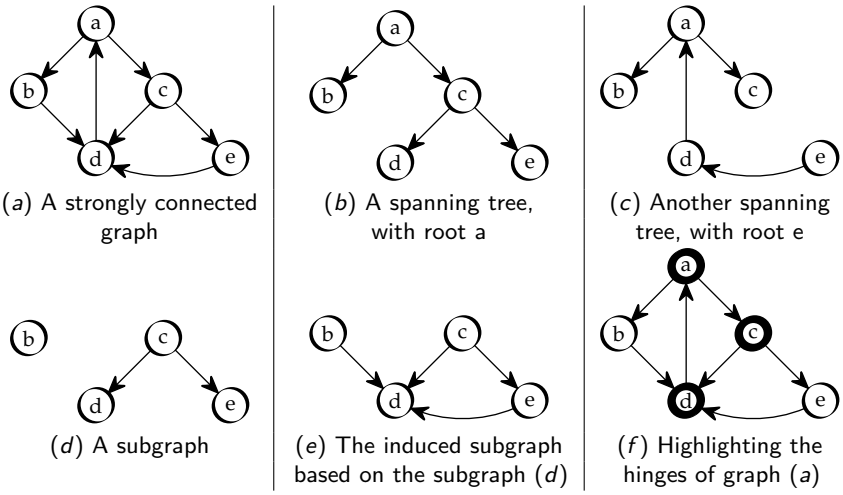


Figure 8.3: A graph (*a*) and various graphs derived from it. The subgraph (*d*) is just some of the arcs and some of the vertices and, as it happens, is not connected. In contrast, an induced subgraph (*e*) contains all relevant arcs. The spanning trees (*b* and *c*) are subgraphs of (*a*).

a subgraph of the device's full graph: in other words, the user knows some, but not all, of the states and actions.

Probably the most important two sorts of subgraphs for designers are so-called spanning subgraphs, which have all the original vertices but not all the arcs, and induced subgraphs, which retain all of the relevant arcs but not all the vertices.

Figure 8.3 (p. 239) shows a simple graph, some subgraphs, two different spanning trees, and an induced subgraph.

▷ Trees and spanning trees are discussed at length in section 8.7 (p. 249) onward later in this chapter.

The figure also shows the hinges (sometimes called articulation vertices) of the original graph. If you delete a hinge, the graph becomes disconnected. If a user does not know about a hinge, they cannot know about the component of the graph on the other side of the hinge. It's therefore crucial that users know about hinges—or that a graph does not contain them, or contains as few as possible.

We'll consider induced subgraphs first. If, from an entire graph for a mobile phone, we pulled out just the vertices corresponding to a submenu of the phone (say, all its language choices), we would expect to find that the induced subgraph was indeed a cycle. In other words, if a user gets down to this language submenu the graph of that menu should be a cycle; otherwise, the user would not be able to get around it easily. In fact, the induced graph of any menu should be a cycle for this sort of device.

8.3.1 Spanning subgraphs

Spanning subgraphs cover all of the original vertices and include enough arcs so that the subgraph is connected. Thus if a user knows a spanning graph of a device, they know everything it can do, but they may not know how to get it to do everything. In particular, if the known spanning graph is not strongly connected, the user will know some things the device can do, but they will sometimes be unable to do some of them—that is, when the device is in a state that is not connected to where they want to go.

What a user knows about a device is of course unlikely to be equivalent to the device: they won't know the device's graph in detail; they may have missed arcs (they don't know that a button does something in some state) or they may have misrouted arcs (they incorrectly think a button does something, when it does something else). However, they must know a subgraph of the device if they are going to be able to use it at all reliably (unless they are playing and don't care whether they remember how to use the device).

A user might rely on a user manual rather than their own understanding of the device in order to use it, in which case, as designers, we should check that or use an automatic (or partly automatic) process that ensures that the user manual represents a subgraph of the system.

▷ In chapter 9, "A framework for design," and chapter 10, "Using the framework," we will talk about generating user manuals further; as a general principle, you can generate smaller user manuals for subgraphs. The user manual may have interestingly different graph theory properties than the interactive device it documents.

8.4 User models

We have described graphs as if they define what a system does. The user's brain is also another system, so graphs define in some sense what the user thinks. In particular, if a device is a graph (it is!) then the user's mental model of how it behaves is also a graph.

As designers, we clearly want the user's model graph to correspond with the actual graph of the device. The user's model should be a subgraph: there may be bits of the device graph that the user does not know, but what they do know should be identical to those bits of the device. That requirement is the same as saying that the user's model is a subgraph.

In practice the user's model is likely to be *almost* a subgraph: the user won't know everything, much of what they know will be correct, but there will be some embellishments. See figure 8.4 (p. 241) for a suggestive pair of user and device models.

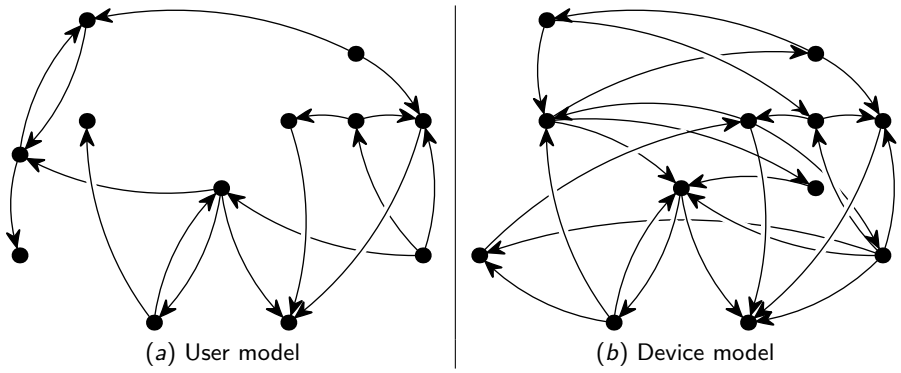


Figure 8.4: An illustrative user model (*a*) is almost a subgraph of some device model (*b*), except that the user thinks there is another state that is connected to the rest of the state space of the device.

8.5 The web analogy

When you start a word processor, your computer shows a screen, which could be simulated by a web site, with the word processor's features shown as little pictures on the web page. When you click on word processor features, you could just as well be clicking on a web link. If you bring up the dialog box to save the file you are working on, the word processor's screen changes: equally, you could imagine clicking on a link and the web bringing you a page that *looked* the same.

Or suppose you were thinking about a mobile phone. A web page could show a picture of the phone, and the keys could be linked to further pages. When you click 1, you could be taken to a page that shows the mobile phone's display changed to show the 1 you've dialed.

▷ In chapter 9, "A framework for design," we show how to make state machines work on the web, and in particular in section 9.5.1 (p. 294) shows how to make a state machine diagram (in fact, an imagemap) work by clicking on its states.

In general, anything—computer program or simulation of a gadget—can be handled as a web site, perhaps an enormous web site. The point is we can talk about the way anything is designed just by considering one type of system and how it is designed. Rather than talk about each sort of interactive system as if it were different, we can see the key interaction issues are shared with everything.

In graph theory terms, a web page is a vertex, and the links written into HTML indicate where arcs go. The label of the arc is the hot text.

Actually, a web page (or a set of web pages) can be converted to a graph in a variety of ways, depending on which conventions we follow. The label could be the hot text, or it could be the name of the HTML link, which the user normally cannot see: `<a href=#linkname>hot text</a>` could be the start of an arc with either "linkname" or "hot text" as its label (or both). Is the HTML text `<a`

Graph theory	*State machines*	*Web sites*	*Hypertext*	*Garden mazes*
vertex	state	page or anchor	node	junction
edge or arc	transition	link	link	path
label	button name	hot text	link text	rarely labeled!

Figure 8.5: Equivalent terms from different fields. Is it ominous or obvious that the same theory works equally well with web sites and mazes?

`href=#linkname>hot text</a>` the initial vertex, then, or is the page containing this text? (It's more useful to make the whole HTML page the vertex; otherwise, the out degree of every vertex would be at most 1.) Similarly, the end vertex is either the text between the tags, `<a name=linkname>target text</a>` or is the page containing the tag.

▷ Section 9.4 (p. 280) shows how to convert a finite state machine into a set of web pages or into a single web page. Either choice can be a faithful representations of graphs.

■ ■ ■

Consider the following syllogism:

- A (big enough) web site can simulate any interactive device or program;
- A web site is a graph;
- Therefore any interactive device is a graph.

There are other ways of putting this, but this way of putting it makes intuitive sense. In fact, as this chapter has made clear, "interactive device" can be much broader than a handheld electronic gadget. Even a maze is an interactive device, where the user's actions are not button pressing but walking.

8.6 Graph properties

When designing a device we will want to check that many things are right or as good as they possibly can be, particularly things having to do with its usability. In general, the questions we want to ask will depend on what we want the device to do and how we want the user to experience it, and how the user is supposed to be able to operate it. Importantly, many things we are concerned about in interaction programming (how fast, how easy to use, how reliable, and how safe) can be expressed in graph theory terms and then precisely checked.

Some people think that we are confused if we say that we can check usability from a graph. Usability depends, they say, on much more than technical properties of graphs! Does the device have the right feel? What do real users really think?

What happens when the device is used in its intended setting, rather than in a laboratory setting or in the designer's office? Usability, they say, is far more.

To avoid the confusion, it's important to distinguish between different sorts of properties and how we use them; no single property on its own can claim to characterize "usability" in its widest sense, but we can nevertheless make some very insightful, and often very clear, assertions.

- If a necessary property is absent, a device won't work correctly. Two very important necessary properties are liveness and safety.
- A safety property says that something (usually dangerous) cannot happen.
- A liveness property says that something (usually desirable) can happen.

We do not need to agree about specific properties. For example, if I am pointing a gun at an aggressor, to me it may be a liveness property that I *can* fire my gun, but to the other person, it may be a safety property that I *cannot*. Yet we'd both agree that the property was important!

- A metric, or measure, says in some sense how hard something is to do (of course, it could measure other factors).
- A metric may not be accurate; it may even be inaccurate in a systematic way. A low bound is a measure that lets us be certain that the correct value is larger; a high bound lets us be certain that the correct value is lower. Generally, our measurements will be off by an unknown factor, for instance, because we don't know whether the user is an athlete and how fast they perform—and then we can only compare measures made the same way relative to one another, rather than make absolute judgments.

Metrics and properties are often mixed up. How well can we achieve a property? Is it true in 65% or 95% of states, say? Or a property could be that a measure achieves or doesn't exceed a certain value. A cash machine might have a safety property that it will never dispense more than a certain amount of money.

In contrast to all these hedges, a property or metric worked out for a graph is quite precise. (Occasionally we deal with such large graphs that we can only guess on the property, but that's not a problem for any properties or graphs that we will consider here.)

We might work out that there are *exactly* 23 arcs between this and that vertex. Whether that 23 means that a user would take 23 seconds or 17 days to get from one to the other, we cannot say. We can guess that if the number was reduced, the user could be faster. We can guess that if the number was enormous, the user might never make it. We can be certain, however, that if we increase the number, even the fastest user would take longer.

All numbers, measures, and properties work like this; in graph theory terms we can be certain of what we mean; in interaction programming terms and usability terms, we need to interpret the properties carefully.

In design, we may find some properties are not as we wish; we then improve the design. For example, we may want to revise a device design so that some things

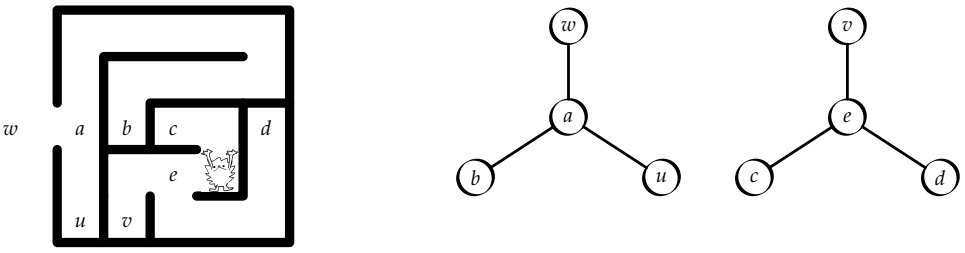


Figure 8.6: The simple maze of figure 8.2 (p. 235) redrawn with an extra wall, cutting off *c*, *d*, and *e* from the rest of the world, *w*. Building this wall in the maze creates two new vertices, *u* and *v*. Representing the new wall in the graph means deleting the old edge between *a* and *e*—the maze now has a disconnected graph.

we've noticed become impossible to do—this is an aspect of debugging, that is removing all actions that lead to device faults. Often, though, when we are debugging, we go about it in an informal way. Graph theory gives us an opportunity to catch at least some design bugs systematically.

Sometimes you may want a property but not want it in a particular design project. A manufacturer may want to sell two versions of a product, a basic one and a version with a value-added feature. The value-added feature should be checked as actually unavailable on the cheaper version of the device but available on the more expensive version. (You would certainly want to check that if a user of the cheaper device accidentally got the device into some "value-added" state that they could get it back again, without returning it to the manufacturer to fix under guarantee!)

How we use graph properties is a matter of judgment. Certainly they can protect us from many silly and some profound design mistakes, and they can help guide us in improving design.

Many of the properties we want a device (or a graph) to have depend on what sort of graph we have chosen to represent the design. Although I will give examples, there are always different contexts; in different contexts, with different assumptions, different properties would be better.

8.6.1 Connectivity

To be able to use a device freely, users must be able to get from any state to any other, and that means that there must be paths in the device's graph from every vertex to every other. Thus a simple usability question has been turned into a simple but equivalent graph theory question.

A graph that is connected has paths from everywhere to everywhere; a graph that is disconnected does not. Compare figure 8.2 (p. 235) with figure 8.6 (p. 244), which is a disconnected maze. For such a simple maze, it is pretty clear that it is impossible to do some tasks, like getting in from the outside world (represented by the vertex *w*) to the center (*c*).

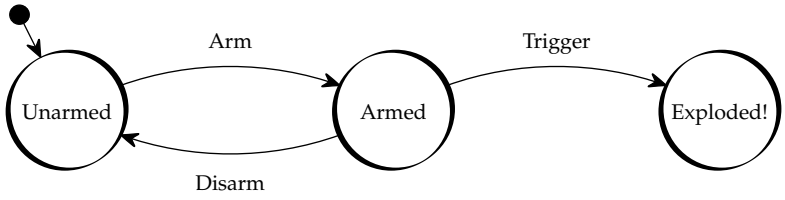


Figure 8.7: A graph that is connected but not strongly connected. A user can go between Unarmed and Armed states without restriction, and from Armed to Exploded, but not back again to Armed (or Unarmed) once the Exploded state has been visited. Compare this graph with figure 9.1 (p. 275), which is a similar, but has an extra arc and is strongly connected.

Most mazes allow you to walk in either direction down the passages; their graphs are undirected. Interactive devices are better represented as directed graphs, in which the notion of connectivity has to be stronger. States may be connected, but are they connected in the right direction? Figure 8.7 (p. 245) shows a directed graph that is connected, but not strongly connected. Every vertex in the graph is connected by a path through edges to every other vertex—but not if you try to follow the edges in the one-way directions indicated by the arrows.

This book is a graph, from which we can extract many interesting subgraphs. For example, figure 8.8 (p. 246) shows the subgraph defined by the explicit linkage between sections. For example, one arc in the figure represents the link from this paragraph to the figure itself. The graph is clearly not strongly connected. It is a set of components, each of which represents a related topic explored in the book. Unlike a maze, this graph is directed: if users (readers of this book) follow a link, they would have a hard time getting back—the links are one way. If the book was converted to a web site so that the links were active links that the readers could click on, the browser would rectify this directedness by making all links reversible. This is a good example of how an abstract structure (in this case, the book's explicit linkage) can be made consistent and easier to use by a program (in this case, a web browser).

It is easy to write a program to check whether a graph is strongly connected. Every interactive device should be strongly connected, unless the model of the device includes irreversible user actions—for example, a fire alarm has a piece of glass protecting its button: once the user has smashed the glass, there is no going back to the standby state. If the answer is that the graph is not strongly connected, you'd want to know where the graph fails to be strongly connected; for these indicate areas that may represent design faults or deliberate features you'd want to check.

▷ Section 9.6 (p. 297) gives one way to check strong connectivity.

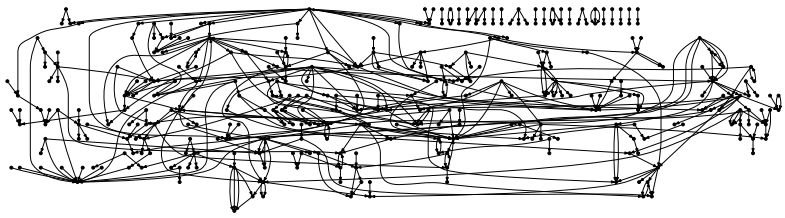


Figure 8.8: This is the graph of all the linkage cross-references in this book. Each section in the book is a vertex, and if a link refers to a section, we draw an arrow from one section to the other. For convenience, we do not show the section numbers (it would make the visualization too detailed) and we do not show isolated vertices. Connected components of this graph are clearly visible, and the small ones represent coherent sub-themes within the book. Of course, if we allowed that each section is connected to its successor, and added linkage like the table of contents and the index, the graph would be more realistic in terms of representing how the user can navigate the book—but harder to interpret! Put another way, if I make this book into a web site, the graph of the web site would contain this graph, shown above, as a subgraph.

8.6.2 Complete graphs and cycles

Graphs can be designed to *be* strongly connected. In particular, the complete graph is one in which there is an arc from every vertex to every other vertex—so it is strongly connected. This corresponds to a device with a button for every pair of states, and the user can get to any state from any other simply by pressing one button. The simple light bulb, with a push on and push off button, has a complete graph of two vertices and two arcs.

In a complete graph there is no requirement that the same button always takes the user to the same state: connectivity and completeness do not care what the arcs are called (that is. the names of the buttons the user has to press), merely that there *is* a connection from every state to every state. A stronger usability question is to ask whether all arcs have the same name as the state they go to: this would clearly make the device even easier to use.

Another strongly connected graph is the cycle or cyclic graph. In a cycle, each state is connected to one other state, and the states are connected in sequence, eventually returning to the first state. Thus you can get from any state to any other state by doing a sequence of actions. For many devices, the consistency would make them easier to use—just keep going and you are *guaranteed* to eventually get anywhere. There is a naming question again: whether all actions have the same name, Next (or ^), say. Repeatedly pressing Next should take the user through all states cyclicly.

It is an interesting contrast that for a complete graph we typically want to check that each button always goes to the same state, whereas for a cycle we want to check that the same button always goes to the *next* state. As we said, many of the

properties we want a device (or a graph) to have depend on what sort of graph we have chosen to represent the design.

We could write a program to find all cycles in a device's graph: typically we would expect to find many cycles, for instance, because the device will have a correction action so the user can undo unwanted menu selections. Often we expect there to be many simple cycles such as menu↔submenu. If we are checking a device design, then, we might first delete all arcs corresponding to correction or undo actions and then check that the remaining cycles consistently use Up (or ∧) and Down (or ∨). The sorts of questions we can pose of a device are endless!

8.6.3 Cliques and independent sets

If each button or action available to the user does what it says, the device will be easy to use—though for some purposes it might be *too* easy to use. The On switch of a light bulb switches it on; the Off switch switches it off; that's the sort of sensible design we want for a light bulb.

In graph theory terms we can spell out this simple property for a device, as follows:

- If a device has a complete graph, then the arc to each state (the button or action getting to that state) can be uniquely labeled with the name of the state. That is, each button does what it says.

To be complete requires exactly as many buttons as states. Unfortunately if you have a device with more states than buttons available, it cannot be as simple to use as this.

However, parts of the device may be complete. A part of a graph, considered on its own, that is complete is called a clique. Even if a graph has several cliques, it may not be possible to name the buttons consistently, since some button names may be used in more than one clique—they'd have to do different things for each clique. Instead, it may be more helpful for a designer to think about the parts of the graph that are *not* cliques. The graph theory concept of independent sets helps here.

An independent set of a graph is a set of vertices taken from the graph such that no pair of the vertices in the set is connected in the original graph. Another way of putting it is that in an independent set, the shortest path getting from anywhere to anywhere takes *at least* two steps.

Or we could say that if the user is "in" an independent set and wants to do anything else within the set they *must* have to do more than just choose the right button, because to do anything else in the set requires having to press at least two buttons. To get around within an independent set, the user has to plan and think beyond just what the next button press will do immediately.

Given the graph of an interactive device, the sizes of the independent sets gives the designer a useful indicator of how hard the device is to use. Traditionally, graph theorists are interested in the maximal independent set (since there are generally lots of them), and the size of this set, which is called the independence number, is a useful metric.

If the independence number is zero, the graph is complete and in principle easy to use (if the buttons are labeled sensibly). The bigger the number, the more difficult the device. Since bigger graphs typically have bigger independence numbers, a more realistic measure for usability is based on the probability that the user has to think, rather than just read and interpret button labels. What we can call the independence probability is an estimate of the probability that the user has the device in an independent set and also wants to get the device to a state in the same set. We could write programs to work this out for any particular graph—it's only an estimate because we don't know precisely which states the user is likely to be in.

▷ Section 10.4.3 (p. 343) gives example program code to work out the independence probability. Chapter 10, "Using the framework," gives many other examples.

▷ Section 9.6.5 (p. 311) assesses how the user can solve the task/action mapping problem—the problem of getting from one state to another efficiently. We will be especially interested in the designer assessing how hard this is to do over all cases that interest the user. As we've seen, for any tasks that involve an independent set, this is a nontrivial problem for the user.

8.6.4 Bridges and hinges

I've emphasized that any normal interactive device must be strongly connected: it must be possible to get from anywhere to anywhere. It makes sense then to talk about how well a graph is connected. In an interactive device, some things are more important than others for the user to know about. In an extreme case, if we do not know what the Off button does, then lots of things may become impossible to do—once stuck, always stuck! In general, missing a little information will disconnect our whole model of the device.

- If deleting a vertex disconnects a graph, the vertex is a hinge.
- If deleting an arc in a graph disconnects the graph, the arc is a bridge. Note that these bridges have nothing to do with the real bridges over the River Pregel—which were (at least in graph theory terms) edges and *not* bridges.

Identifying bridges and hinge vertices in a graph is routine. Every bridge and hinge represents a piece of critical information: without it, the user cannot (or does not know how to) access any states on the other side of the bridge or hinge.

If the device is supposed to be easy to use, then it should have few hinges, or at least, few states hiding behind hinges. On the other hand, if the device is safety-critical—if it can do dangerous things—then it is probably best if these dangerous states are safely protected by bridges, perhaps even chains of bridges.

If a user does not know about a hinge, then it follows that they cannot get to some states (they might be able to proceed by accident). Having found a hinge, the graph can be separated into at least two component subgraphs. One of these

components will include the standby or start state, which we'll assume the user knows about. The question is, what's in the other components that the user cannot get to except via the hinge? If these components contain states that are important, then the user *must* know about the hinge.

It may be appropriate to make the hinge (if the graph has one hinge) into the standby state (or the home page for a web site) or otherwise redesign the device so that hinges are close to standby and therefore easier to find and more familiar through use. Conversely, if a device has more than one hinge, as it can't have two standby states, this may well be a design problem the designer should know about.

▷ Section 10.1 (p. 325) introduces the farmer's problem and uses it to give further examples of strongly connected components and how they relate to design.

Some graphs have no hinges. The connectivity of a graph is defined as the smallest number of vertices you'd have to delete to disconnect the graph. In particular, if deleting just one vertex disconnects the graph, that vertex is a hinge. If you need to delete two vertices to disconnect a graph, then the user has two routes into the states on the other side of them.

If the connectivity of the graph is two, the user is safer, but if they do not know about at least one of the two vertices, so again part of the graph will be unreachable for them, except by accident. Can we design to ensure, as far as possible, that the user knows about at least one of these vertices? As the connectivity of a graph increases, it becomes less dependent on critical knowledge—the user has more ways of doing anything.

As a conscientious designer, you will want to (somehow) make sure the user understands the importance of every hinge in a graph when they are using the device. Perhaps the device should flash something, or beep. Or perhaps a button should flash when the device is in a state that makes buttons correspond to a bridge.

Graph theory gives us ways of evaluating interesting usability issues and giving them precise measures, which can then be used to guide modifying designs and try to improve them against the measures. Graph theory also suggests ways of finding things that can be improved without changing the structure of the device. We could either eliminate bridges from a design, or we could highlight them so a user is more aware of their importance. Or we could write more prominently about them in the user manual.

8.7 Trees

Graphs might seem pretty trivial things on the grand intellectual landscape—just arcs and edges—but trees, which are even simpler and therefore easier to dismiss, are in fact *very* useful.

Trees are of course familiar in the countryside, and sometimes help form the walls of mazes. Ordinary trees can also be represented as graphs; in fact, a tree-

like graph is so common that tree is the technical term in graph theory. A tree is a particular sort of graph that, well, looks like a tree.

For some reason, as shown in figure 8.9 (p. 251), we usually draw trees upside down, rather looking like the underground roots of a tree without the tree itself. Indeed, we use the terms "above," "beneath," "top down," and "bottom up" assuming trees have their roots at the top.

A collection of trees is, unsurprisingly, called a forest. Some people call the arcs (or edges) in trees their branches, and the ends of branches (that don't go on to other branches) are called leaves. The ends of branches, whether they are leaves or not, are called nodes, just as an alternative to the word vertex.

The root of a tree is a special node with no arcs going to it. We can define leaves similarly: a leaf is a node with exactly one arc going to it and none from it.

A tree is connected, but as there is only one branch connecting any two adjacent nodes and there are no cycles (because a tree does not grow back on itself), a tree cannot be strongly connected. Trees have the property that there is *exactly* one path from the root to any other vertex; if the tree was undirected (so you could go back on the arcs) then there'd be exactly one path between *any* two nodes. Put another way, every vertex in a tree is a hinge and every arc a bridge.

Trees are very useful information structuring tools and have all sorts of uses. They are often used in family relationships and genealogy. Branches can represent the relation "child of," and inevitably leaves or nodes of a tree at the same level immediately below a node are then called siblings. This sort of tree shows the descendants of the person named at the root of the tree. Or a genealogical tree can be used the other way round, so the arrows represent "parent of." In this case, you would be at the root, and your parents would be one level *down* from you, and your oldest ancestors you know would be at the leaves of your tree.

Trees are very often used to represent web sites; the root is the home page and the leaves are pages where the user does things (like buy a book, if each leaf is a book description); all the other nodes are then basically navigational, but they might also have incidental information on them.

In this book, we need directed trees, where each branch points away from the root. Trees are ubiquitous in user interface design, because almost every device presents the user with choices—menus being the obvious example. For a user interface, the root of the tree would probably be "standby" or "off" and the other nodes might be functions the device can do or menus of further choices. Users might think that functions are "in" menus but in fact the *choices* are in the menus and the *functions* are leaves one arc below the menus.

A problem for a user interface designer is that it is tempting (and, indeed, often sensible) to draw a tree to guide the design of the system. Trees are simple and therefore good for conceptualizing a device. But because a tree is not strongly connected, designers may forget about the importance of strong connectivity. One may end up with a device that has a tree-like user interface that appears to be simple and elegant, but which has some weak spots that are not strongly connected.

More precisely, a designer trying to design a device as a tree should think of it as a subgraph. If this subgraph is a tree that includes every state of the device, they

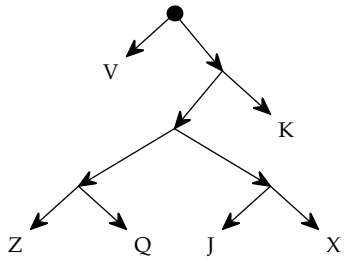


Figure 8.9: This is a subtree of figure 5.5 (p. 129), which was a Huffman code for the complete 26-letter alphabet. The root of the (directed) tree is shown as a black disk; the leaves are shown as letters; the branches are arrows. Either end of an arrow is a node; if no arrow points *to* a node, it is a root; if no arrow points *from* a node, it is a leaf. To be a tree, there is exactly one path from the root to any leaf.

are designing a spanning tree. A spanning tree guarantees exactly one route from its root to every other vertex. Users who know a spanning tree for the device, provided the device starts off at the root state, can do anything with the device even if they don't know any more about it. In general, the root will be some obvious state, like standby. Typically we would then require every state to have some obvious action that gets back to standby—this would not be part of the spanning tree, but it would ensure that users can always get back to the root easily and thence to any other state by following paths in the tree. In fact, in this case, users would need to know meaning of some key like Reset, or else they would not be able to navigate the interface.

8.7.1 Trees are for nesting

Imagine you are using a gadget and it is in standby. You want to access one of the device's functions. If you know how to do it, you will press buttons and follow a short path through the graph that represents the gadget until you get to the state you want. If you worked out the shortest paths to get to every state from standby and drew them on your state diagram, you would have drawn a tree with its root at standby. If, further, we make the assumption that a path to a state never activates any other state we are interested in, then all the states we are interested in will be leaves of the tree.

Any device (precisely, the graph of its finite state machine) contains trees, which represent the best way of using the device starting from a key state, such as standby. In graph theory terms, any strongly connected graph has at least one subgraph that is a directed tree. To design a device, we first design a simple tree, then we can add more arcs to give the user more choices in how to achieve things with the device.

▷ Subgraphs are defined in section 8.3 (p. 238).

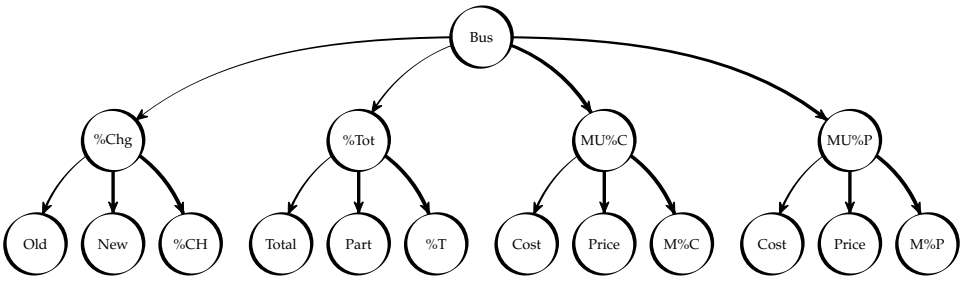


Figure 8.10: Part of the tree-based menu structure for the Hewlett-Packard 17BII handheld financial calculator. Cost and Price are both shared under two different headings, so it is not a proper tree (it would be a semi-lattice). The terms are HP's—they appear exactly the same, but in block capitals on the calculator display: "Bus" means business, "%Chg" means percent change, "MU%C" means markup as percentage of cost, and so on.

This book is a graph that contains lots of trees. For example, the book's index is the root of a tree of ideas and people's names; each page number in the index is a branch leading to a page, which is the leaf. Or the table of contents is a root of another tree, and each chapter the first node down from it, with sections being children of the chapter nodes ... down to paragraphs, words, and letters finally as the leaves.

All these trees structure the book in different ways, but none of them are any good for reading the book. For reading, there are two further sorts of arcs added: one, implicit, is that as you read you read on to the next leaf without thinking—you read the book linearly; the second sort of arc is the linkage and cross referencing that encourage you to read nonsequentially.

▷ For a brief history of the inventions that make reading easier, see section 4.2 (p. 96).

The ideal plan is not as easy as it seems. Figure 8.10 (p. 252) shows a subgraph of the structure of the Hewlett-Packard 17BII handheld financial business calculator. The structure shown is a tree, but some items are actually shared between branches. Thus the simple-looking tree structure in figure 8.10 is misleading, since it looks like a tree only because the repetition of COST and PRICE is not visually obvious. The design of the 17BII therefore makes the user's choices for COST and PRICE permissive.

▷ The advantages of permissiveness are discussed in section 5.2 (p. 134).

Drawing the structure pedantically, so that each item had exactly one box, would in general make lines cross, and the diagram would become cluttered and unhelpful. A designer would be unlikely to design it looking as bad as this, which would unfortunately encourage them to make it a strict tree.

Merely drawing trees as a visual aid in the design process tends to lead designers into making user interfaces that are too strict. You almost always need to add

Box 8.2 Trees versus DAGs You can't go round in loops in directed or undirected sorts of tree, so they are said to be *acyclic* graphs. Acyclic graphs are useful—they guarantee you can't get into loops. You may not get where you wanted to go, but you will eventually get somewhere! In contrast, a user (or a computer program) can get stuck going around a loop in a cyclic graph forever.

Directed trees are a sort of directed acyclic graphs or DAG. But DAGs allow more than one path between the vertices, so they are more general than trees. Many DAGs are trees, and all directed trees are DAGs, but not all DAGs are trees. A DAG might have several roots—vertices that have no others pointing to them—whereas a tree can only have one root.

While you might design a device as a DAG or tree, very few interactive devices would be satisfactory as DAGs, because they do never allow the user to go back to previous states. Typically, then, you may *design* a device as a DAG, but use a program to systematically add more arcs to it to make it easier to *use*.

further arcs (and that's best done systematically by program), and great care must be taken that repeated states are exactly the same state, rather than being duplicates that may not be exactly the same. For example, the state Cost in figure 8.10 may or may not be exactly the same state, though it has the same name, which suggests to the user it *is* the same state.

8.7.2 A city is not a tree

Although trees are clear and easy to draw and, as the last section emphasized, trees are subgraphs of any design, and they can be misleading design tools. Trees encourage designers to slide into bad design, an insight first spotted by Chris Alexander in the context of town planning.

Chris Alexander is an architect, famous among programmers for introducing pattern languages, but also widely known for his views in architecture. In his classic discussion of the design of towns and cities he argues that town planners too easily lay out a city as a tree.

In a tree-structured design, the industrial area is over here, the housing is over there, and the shopping is somewhere else. A tree-structured design makes it conceptually very easy for the designers. While the designer keeps the design concepts, "housing" and "shopping" and so on, separate in their minds, they have a lot less to worry about. Indeed, it's called "separation of concerns" and is a design principle that encourages you to avoid getting features mixed up.

Unfortunately if a town is designed as a tree, the shopping area is dead at night because nobody lives there, and the housing area is dead during the day, because everybody is at work or school. So Alexander argues that a well-designed place mixes the housing, shopping, and work regions. Mixing means that shops do not appear in one place, workplaces are not in one segregated area, and so on. A good city, then, is not a tree—most things (housing, shops, doctors) can be found in several places in several different ways. A good city is permissive.

Indeed, if we push Alexander's thought to the limit, even in a highly regimented tree-structured city it is hard to imagine that there would be only *one* place where a user could find a newspaper—people find duplication of resources (whether newspapers or user interface features) helpful. But trees do not allow for any duplication: everything has to be in its place as a single leaf.

Figure 8.11 (p. 255) shows an alternative, but equivalent, view of the HP calculator tree drawn in figure 8.10 (p. 252). Figure 8.11 can be imagined as viewing a tree seeing the menu choices as enveloping circles: each menu encloses all of its submenus. More formally, this alternative view is a tree represented as a Venn diagram, as a collection of sets.

▷ Venn diagrams were developed by the British logician John Venn in 1880 from a simpler and much earlier visual formalism of Euler's, Euler circles (this is the same Euler who invented graphs). Statecharts combine the two techniques, and were reviewed in section 7.6 (p. 217).

Because the diagram represents an interactive device, each set (drawn here as a circle) appears to the user as a menu, and its elements are the submenus (or the functions) of the menu. In a tree the sets that define it do not overlap. In fact, *any* set of nonoverlapping sets defines a tree (or a forest of trees if there is no overall enclosing set to define a single root).

As we said above, the 17BII is not strictly structured as a tree: the MU%P and MU%C sets *should* be drawn with some overlap, with their intersection containing COST and PRICE. In short, drawing simple diagrams—here, trees and Venn diagrams corresponding to trees—tends to encourage design oversights.

Just as a city designer finds a tree convenient, a designer of interactive devices finds a tree easy as a structuring device because each function or purpose of a device goes in one sensible place. For a mobile phone, all the network settings would go under Network; for a city, all the houses would go in Living Quarters. Once you've decided to have an area called Language, the rest of the design fits into place.

But this means that to use a tree, the user must know the designer's classification. My mobile phone has menus called Extras, Settings, and Profiles. Where would I find the alarm clock feature? Is it an extra? Is it a setting? Or is it something to do with my profile? Just like a city, if functions, like alarm settings, appear more than once in different sections (particularly in different forms or aliases) they will be easier to find. I could find the alarm clock under Extras or Settings and not need to choose "the" right one. But if the mobile phone is of this easier-to-use structure, it will not be a tree.

8.7.3 Getting around trees

Trees provide a single, short path from the root to every leaf. Thus, as interactive devices, trees ensure that the user has a short route from standby to any function—provided they know the route.

However, if a user does not know the right route through the tree, they are likely to get lost and then they need to go back, probably all the way back to the root. But

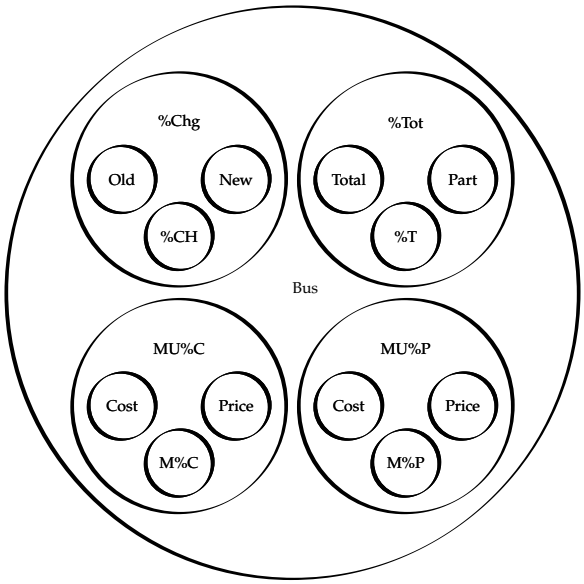


Figure 8.11: Venn diagrams are nested nonoverlapping circles that represent trees. A Venn diagram is used here to represent the command structure of the HP 17BII calculator, which was also shown in figure 8.10 (p. 252) as a tree. As the text explains, this drawing is subtly misleading—it looks simpler than the device really is.

directed trees are not designed for going back. Or if a user wants to explore a tree to see what it has in it, this again is not a simple get-from-root-to-leaf procedure.

In short, trees are not sufficient for all but the simplest user interfaces; even if functions (leaves) are repeated to make a tree more permissive and easier to use, a user will still sometimes want to navigate the structure in an essentially a non-treelike way.

Unfortunately, searching a tree efficiently is quite complex. The user has to go into each submenu, search it, then come out of the submenu and search the next submenu. It is very easy for a user to make mistakes, go around some submenus more than once, and (worse) come out of a submenu prematurely and miss out altogether a subsubmenu that might have what they are looking for somewhere in it. Once a user makes a mistake, they may "panic" and lose track of where they have got to, even if it wasn't too hard before getting lost in panic!

For a user who does not know how a device tree works, or what classification scheme a designer used for the tree, a linear list would be far better. In fact, a cycle (a list that joins up at the end to start again) would probably be better. You can start on a cycle anywhere and eventually get everywhere, whereas with a linear list, you have to know where the starting point is and you get stuck at the end. Whichever you have, there is no wasted time in learning how the list is organized; for users who learn how it is organized, they are guaranteed that a simple scroll

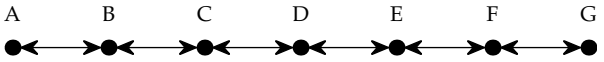


Figure 8.12: A linear list for seven commands. The user interface merely requires buttons equivalent to [Next], [Previous] and [Select].

(say, repeatedly pressing [∨]) will get through everything in the entire list. With a list, a user will inevitably find whatever they want (provided it is available).

In short, we often want to design a device that is both a tree *and* a cycle (or, more precisely, contains two subgraphs, a tree and a cycle, whose vertices are the device functions). It is not difficult to build a design tool that allows the designer to see the device as a tree but which systematically inserts all the cycle links in—to help users in a consistent way, without distracting the designer with the added complexity of extra permissiveness in the final design.

▷ Section 5.5.5 (p. 153) compares different structures (including the manufacturer's original tree) for searching and access times for a fax machine.

8.7.4 Balanced and unbalanced trees

Trees are very efficient for searching, *provided* they are organized appropriately. For simplicity, suppose the device has seven functions called A, B, C, D, E, F, G—instead of spelling out their full names, like "send text message" or "rewind" or whatever functions your favorite device does.

The user could have an [Up] and a [Down] button and simply scroll through the commands. Figure 8.12 (p. 256) shows the graph of this device. Here there are 7 commands, and on average the user would need to search half way, taking 4 button presses to find what they want.

The average is 4 because if we start at A, then selecting A (which is done 1/7 of the time) can be done in one press, [Select]; selecting B would take 2 presses, namely, [Next], [Select]; ... up to selecting G, which would take 7 presses. The average of $1, 2, \ldots, 7$ is 4. The cycle, figure 8.13 (p. 257), does better with an average of 2.7. Depending on the usage we expect, a cycle may or may not be an improvement, however: (for a sufficiently large cycle) will some users go round and around endlessly and be stuck, or will knowledgeable users always be able to take advantage of the almost-halved average distances?

A binary tree, such as the one shown in figure 8.14 (p. 257), can do better. Starting from the root (an action that might take a key press we are not counting here) it takes 1 step to activate the function D (which the user will do 1/7 of the time on average), by simply doing [Select], or 2 steps to select either B or F (by doing [Less], say, then [Select]), or 3 steps to select any one of A, C, E, or G. So the average is $1/7 + 2 \times 2/7 + 3 \times 4/7$, or 2.3.

Of course, to gain the benefit (which gets better the more commands there are in the tree) the user must use the right systematic way to find commands. That is,

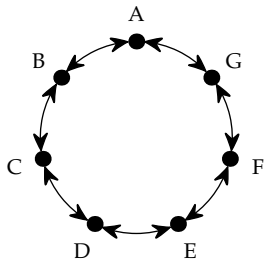


Figure 8.13: A cycle for seven commands. Whether this is better than a linear list depends on whether A and G are related or should be kept separate, and whether it matters that the user may not notice they are going round in circles indefinitely—particularly if the cycle is very big and the user doesn't remember where it starts. The buttons or commands for this user interface are the same as required for a linear list.

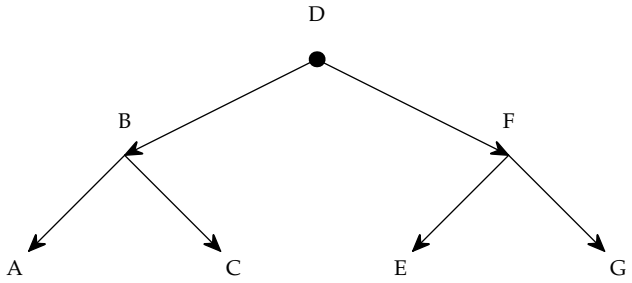


Figure 8.14: A balanced binary tree, with 7 nodes. Imagine the user wishes to select one of the 7 lettered commands, A–G, and starts from the root of the tree.

if the display shows a command X, and if this is what the user wants, they should stop. If not they should press [Before] (or whatever it is called) if what they are after is earlier in the alphabet than X; otherwise they should press [Later].

Unfortunately, there are trees that, though they follow the same rules, are much less efficient. One is shown in figure 8.15 (p. 258). Now the average cost to find a function is $(1 + 2 \times 2 + 3 \times 2 + 4 \times 2)/7 = 19/7 = 2.7$, which is a bit worse—but the more functions there are, the worse it gets. For instance, if we had 127 functions, it would be worse by 33 instead of 6. The interesting thing is that both trees work exactly the same way, and (apart from the time it takes) a user cannot tell between them.

Perhaps a designer can't either. That would be very worrying. There are an exponential number of trees, but there is only one balanced tree (if it's full). The chances that a designer hits on the best tree by chance is negligible. Unless the designer uses design aids or tools, they are very unlikely to make an efficient tree by designing it by hand—it is too much work.

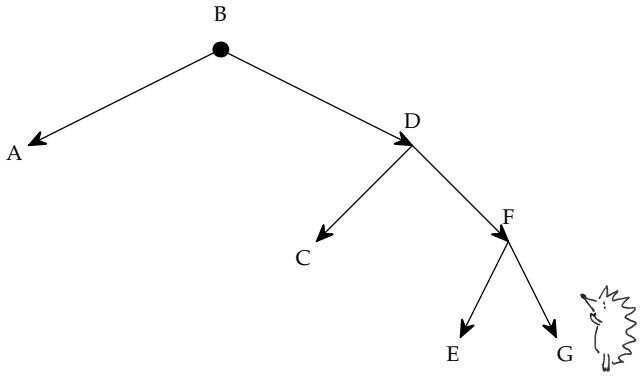


Figure 8.15: An unbalanced binary tree, with 7 nodes. The user's strategy for using this inefficient tree is the same as for using the efficient balanced tree of figure 8.14 (previous page).

In these two examples of balanced and unbalanced trees, I've assumed that the tree is a binary tree, with two choices at every node. If we increase the number of choices, the user can get to where they want faster. The "hidden" cost is that showing the user more choices requires a bigger screen, and as the screen gets bigger, the user will of course take longer to read it—and perhaps not read the occasional extra lines sometimes needed, but which don't fit in.

The moral is that when you use trees, balanced trees are much more efficient—and for some applications, such as alphabetic searching, it makes no difference to the user—other than being faster.

We worked out average costs assuming that everything (using any of the 7 commands or functions of the device) was equally likely. If functions are not equally likely for whatever the user is doing, then a Huffman tree will give better average performance than a balanced tree; in fact, if all the probabilities are the same, the optimal Huffman tree turns out to be a balanced tree. There is one proviso to this comparison: when I introduced Huffman trees, the functionality was at the leaves, and in our discussion here, we have functionality of the device both at leaves and at nodes interior to the tree.

▷ Huffman trees are discussed in section 5.1.3 (p. 126).

When we used Huffman trees for the functions of the Nokia handset, the functions of the handset device were at the leaves of the tree, but submenus (like "Messages") were at interior nodes. With this design, the user would never want to *use* the device's Messages function, because it isn't a function—it is merely a menu providing all the different sorts of message function.

8.8 User manuals

Books contain subgraphs that are trees. A book has chapters, and within chapters, sections, and within sections subsections. The book itself (or perhaps its cover) can be considered the root of a tree, with branches off to the chapters, then each chapter has branches to its sections. Some of those sections may be leaves, and some sections will have further branches off into subsections. If you want to pursue the structure, you can continue down into paragraphs, diagrams, tables, and so on. In fact, this is exactly what XML does: it provides a tree-nesting structure for documents.

User manuals are books of a sort, so they are trees in the same sense. Devices, or subgraphs of devices, are trees too. Both are trees from the vantage point of graph theory. Can we do anything useful with this correspondence?

Since interactive systems are graphs and web sites are graphs, we can easily build a simulation of a device by using a web site. As the user clicks links in the web site, they go to other pages. This is the same as if the user clicked buttons and went to other states of the device.

Web sites are conventionally used more to provide information than to simulate gadgets. The original term, before "web" was suggested, was hypertext, then thought of as a collection of mostly textual pages that were linked together in a more flexible way than bound in a book, which forces them to be ordered sequentially. It follows that the underlying graph idea can also be used to generate a user manual. To do this we first must represent the device as a web, and then we will see how to get the web or hypertext into a conventional written form.

▷ Programming manuals is covered in sections 9.4 (p. 280) and 11.5 (p. 392).

To generate a basic user manual as a web site is easy: instead of taking care to simulate the device and show pictures of what it looks like in each state, we show manual text describing each state. Instead of having hot text called Operate (or whatever) that links to other pages, we write text that says, "If you press Operate then such-and-such happens." If we think of the manual as an interactive web, then pressing Operate in this sentence can "do" whatever it says and take the user to another part of the manual. In a written, paper, version of the manual, the text that would have been shown on another web page can be written out in the following section.

How to do this well raises several interesting issues, not least because there is not a single, best way to represent devices as manuals. Conventional paper manuals are trees in more than one sense: they are made out of paper derived from trees of course, but they are also made up from sections, containing subsections, containing sub-subsections and paragraphs of text—they are trees in the graph theory sense of the word too. But the device we are trying to write the manual for is not usually *just* a tree; it will be an arbitrary graph, and any sensible interactive device will have loops in it (it will typically be strongly connected so it must have at least one loop). Since no tree has loops, we have to think about ways of getting the best trees out of graphs in order to start making manuals.

Or perhaps we might write user manuals on continuous loops of paper, which aren't trees, or only represent them as web sites, which, being arbitrary graphs, can directly represent devices.

8.8.1 Spanning trees

A spanning tree is a tree that uses a graph's original edges to include all its vertices (in contrast to a subtree, which is merely a tree including *some* of the vertices). To make a user manual, then, we could do worse than look for a spanning tree of the device and then format it as a readable manual, for instance with its root as the table of contents branching out into chapters.

Unfortunately there are lots and lots of spanning trees for most graphs, and the chances of finding a good one by chance is slim. We need to have an idea of which one is best. One way to do this is to score everything that could go in a manual with how interesting or insightful it is. For example, to repeatedly tell the user that pressing Off switches the device off is not very informative. So we could score any arc ending in Off low. Many music devices are intended to play music, so any arc that ends in playing music might be scored high.

We want to find the spanning tree with the maximum score, which we anticipate will be of most use or most interest to the user. Such a tree is a maximal spanning tree, and it has the highest cost or score of all spanning trees.

There are many ways to find spanning trees: the most intuitive is to start drawing a tree over the existing graph and make sure you never join up back to a vertex that is already part of the tree (otherwise you would create a cycle)—this is called Prim's algorithm. Another approach is to find any cycle in the graph and delete an arc, provided that deleting that arc does not split the graph into two isolated components—this is Kruskal's algorithm. Both of these approaches can easily be modified (and often are) to choose the highest scoring arcs as they build the trees; finally they result in maximal spanning trees.

Rather than maximizing scores, if we define the arc costs as, say, the length of English it would take us to explain something to a user, then we can set a program to look for a spanning tree that corresponds to the shortest user manual we can generate. Of course, our estimate of the length of English will be approximate, based on some rules of thumb; it'll probably be most easily measured by generating very bad machine-generated English, and simply counting the number of words needed.

▷ We discuss the importance of short, minimal manuals in many places in this book; see also section 11.6.1 (p. 397).

If we improve our English-generating programs, we would be able to get more informative scores than mere length measurements; we could also measure readability or count verbs or lengths of sentences. If sentence length, for example, is an important measure for our users or what they will be doing, we could design our English generators to produce short sentences, and we then choose spanning trees that reduce total sentence length, ending up with better manuals.

Box 8.3 Computers in 21 days Visit a book shop and you'll be spoiled for choice with all the books explaining computers for people like you. Isn't it reassuring to discover that you are not alone and have joined a great band of people who find computers fundamentally difficult?

Using computers requires lots of trivial knowledge, like when to press the F1 key. That's why those self-help books are so thick. Certainly, nothing works unless you know the right trick. When you do know something, it seems so simple. It is then easy to slip into thinking that you must have been stupid for not knowing in the first place.

What if all those people who wrote books on how to use computers talked to the manufacturers who made the difficulties? One of my hobbies is to go through manuals, to find instructions to tell us how to cope with quirks that need not have been there in the first place. When a manual says, "make sure such-and-such," I ask why wasn't the computer designed to make sure for you? Almost all problems with computers are easily avoided by proper design, by manufacturers doing a little thinking first. What are computers for if they can't do obvious things?

Take any other modern product. You can buy a car. You don't get thick manuals telling you how to stop the wheels from falling off. Wheels don't fall off on their own. In comparison, most computers are totally unsatisfactory: you get lots of "wheels" that fall off when you are not watching.

Computers are badly designed because manufacturers can make money without trying any harder. Yet most people say computers are truly wonderful! Before you know it, you too will be buying upgrades, more RAM, and some training books and when you've spent another thousand dollars you'll agree that they *are* wonderful. It's not you who benefits from this, but consumerism that likes to keep you dependent, believing that you are responsible for fixing the computer's problems with more of your money.

Since there are so many spanning trees you can get from even modest graphs, the designer and technical author need all the help they can get. If they choose a user manual structure on their own—any spanning tree—there's little chance that it will be optimal or even nearly optimal.

Even a simple design tool would be a great help. Prim's algorithm has the nice advantage that it can be used to extend an existing or draft manual. I envisage a designer writing a basic manual (which is of course a tree but may not be a spanning tree), then Prim's algorithm will provide suggestions for extending the tree to make it more like a spanning tree—but suggesting a good extension by however we are measuring "good."

A designer would like to be able to assess how easy a device is to use. But we can produce user manuals automatically (or semiautomatically), and there are now more usability measures we can get by measuring the manuals rather than the devices directly. The longer a user manual is, for instance, certainly the longer a user will take to read it. In fact, the longer a user manual has to be to correctly describe a gadget, the harder the device must be to learn to use. We could automatically generate a user manual, and measure its length. Then if we can modify the design and reduce the length of the manual, we have in principle an easier-to-

learn device design. The depth of nesting in the user manual too is another useful measure of complexity. It corresponds to how hard a particular feature is to learn.

To take advantage of these ideas we have to have design tools that can generate user manuals from device specifications. Otherwise, we would make a small change and have to wait until the technical authors had rewritten the manual! A designer needs to be able to experiment with the specification of a device and see whether the manual gets longer or shorter. If this can be done fast enough, as it can be by drafting rough manuals using automatic design tools, then designers can very quickly optimize designs to make them easier to learn or use (depending on what measures they are using).

Generally, better manuals are indeed shorter, but of course there are exceptions. A manual for a gun would have a lot of safety instructions, and blindly removing the safety features so that a manual does not have to discuss them (and would therefore be shorter) would be counterproductive. Different sorts of features should be flagged as having different weights in the calculation of the "length" of the manual. Possibly the more safety discussion, the better—if so, for such a device, text on safety features would have a negative weight; thus the more safety-related text the shorter the manual appears to be. Imaginative designers will be able to come up with more sophisticated approaches that can be programmed into tools.

A technical author can tidy up the automatically generated English. Provided we have a way of keeping track of what the technical author has rewritten, text never need be rewritten unless the automatically generated text for the affected parts changes. Thus if we have automatic tools to help with manual generation from a device specification, we can do something very useful: when the device specification changes, (most) effort put into writing a clear manual need not be lost. Furthermore, we can *deliberately* change the device specification to get a better manual. If our user manual-generating program can quickly give an estimate of the complexity score of the user manual, then it would be worthwhile for the designer to experimentally mess around with various alternative designs to find the one with the clearest, briefest explanation.

8.8.2 Ordered trees

All the trees we have discussed to this point have been ordered trees. As well as being either directed or undirected, a tree can be ordered or unordered.

Just as adding arrows to an undirected tree involves a choice—you have to choose the root, then every arrow direction flows away from the root—ordering a tree involves choice. Every choice one makes in the structure of a system represents a design choice, and if it affects the user of the system then it is an important interaction programming issue. Tree ordering is one such issue.

As figure 8.8.2 (p. 263) indicates, in drawing a tree, the designer must make choices to order the children of every node, placing them left-to-right or in some other order across the page.

The previous section discussed how to construct a user manual as a spanning tree. The spanning tree itself needs ordering before it can be printed—at least

Figure 8.16: One or two directed trees? If these two trees are unordered trees, they are the same (rather, they are different representations of the same unordered tree); if they are ordered trees, they are two different trees. As ordered trees, the trees differ in the order of the two children d and e.

assuming it's not a degenerate spanning tree in which every node has exactly one child, stretched out as list, as in figure 8.12 (p. 256).

How do you order the tree in the order that suits the reader? The best orders depend heavily on the application, but these three ideas can be used separately or in combination:

- The user or a technical author—any expert in reading—gives hints on ordering the sections of the manual. This must be first; this must come before that; that must come after this.
- Flag every mention of a concept in the manual; flag *definitions* and *uses* of concepts. Then order the manual so that definitions precede uses. That way the user gets to read the section that defines what a dongle is before reading sections that say what to do with them.
- If, in addition to being a tree, the manual contains cross references (as this book does), we minimize the total length of cross reference links. If we strongly prefer cross references of length one, that is which go to the previous or next section, then we can remove them from the document since the reader can easily read the next section without being told to by a cross reference!

All of these rules (and no doubt others will occur to you) have corresponding algorithms for implementation.

The first two can be combined; they require topological sorting, which is a standard and straightforward algorithm. Topological sorting may, on occasion, fail to find an ordering that satisfies all the requirements.

For example, the standard dictionary joke that badminton is played with shuttlecocks and shuttlecocks are things used in the game of badminton is a case of two sections both defining words the other uses. There is no ordering that puts definitions before uses. Topological sort will point this out; the designer or technical author will then have to make a decision that one or other rule has to be relaxed—or perhaps the document rewritten so that the problem does not arise. To resolve the dictionary paradox, we need only rewrite the definition of badminton to include its own definition of shuttlecocks; we could then optionally simplify the definition of shuttlecock to the terse "see badminton."

Doubtless, dictionary writers either spend sleepless nights with worry, or they use computers to help them spot problems automatically.

The last of the three ideas for ordering trees is more obscure and requires the so-called jump minimization algorithm, because the problem is the same as reducing the length of jumps in a program with goto statements (which correspond, for the program, to our cross references). A variation of jump minimization is bandwidth minimization, which tries to minimize the maximum length of any cross reference, rather than the total length (which of course might mean that some cross references span great distances).

If the user manual was split into two volumes, an introductory overview leaflet and a detailed in-depth reference manual, it might be wise to minimize the number of cross references from the overview to the reference manual, or we might want to make the reference manual self-contained, with no references to the overview.

There are many choices in ordering a tree. The point is that we can be precise about the rules we wish to apply or enforce to make the user's use of the manual (or device) easier, and we can use algorithms to automatically generate the best results—or to point out to us any problems or inconsistencies in our requirements. There is a key advantage to doing this (which I repeat in many ways throughout this book): if we decide to modify the design in any way, we *automatically* generate new manuals without any further effort.

In contrast, if we worked conventionally, even a minor design change could do untold damage to a carefully written user manual (or any other product of the design process)—and that potential problem would mean we don't make improvements because of the knock-on costs, or that we don't have accurate manuals, or that we write the manuals at the last possible minute and don't thoroughly check them. Very likely, when the manuals are written late, we would not read them ourselves and get no further insights into how to improve the design (some design problems will stand out in awkwardly written sections of the manual).

8.9 Small worlds

I've described graphs and graph properties as if graphs are "just there," as if they are static structures that just happened to be so. A device has a particular design, represented by a graph, and that may be analyzed. Is it strongly connected or not? What is its Chinese postman tour? What user manual might we be able to generate for it?

We haven't yet thought about *how* to get the graphs in the first place. Most graphs in real life grow; they don't just appear out of nowhere. This year's device, and its underlying graph structure, is going to be based on last year's structure: so the device graph for this year will be last year's plus or minus a few more features—a few more states and a few more arcs, and some bug fixes. Device graphs thus have a history, and how they are designed depends a lot on that history.

Suppose the designer wants to introduce new features for next year's model. These additions will join to the old device graph, which probably won't be changed

much, because it would be too much trouble to redesign what is already known to work.

Rather than being created out of nothing, a device graph grows over time. Iterative design occurs for new models based on previous years' models, but even new devices will have had some design iteration: when design started, there would have been some earlier sketch design, and that design then grew and was elaborated over a period of time, mostly by being extended with new features as the designers or users think of things to add.

When graphs grow, they tend to have special and very interesting properties. In technical terms, these graphs are called small world or scale-free, and the popular, well-connected vertices are called hubs. Here are two familiar examples:

The world wide web When I create a web page, I tend to link it up to sites I am interested in and that are popular. Of course, I'm unlikely to link to unpopular sites because I probably won't know about them. In turn, if my page is popular, lots of people will link to me. Again, as more people get to know about and link their pages to my page, then my site becomes even more popular, and even more people link to me. Popular web pages quickly become very popular. Hopefully, my page will end up being a hub.

The airport network There is already a network of airports and rules about where airplanes fly. If I want to build a new airport, I'm going want routes approved to somewhere like Los Angeles or Heathrow, because they are popular airports, and if I can connect to one, I'll make my airport far more popular. Heathrow and Los Angeles are a hubs in the airport network.

If a graph is a small world, the average distance between its vertices will be surprisingly low. In an ordinary random graph, most vertices have about the same number of edges, and there are no shortcuts from anywhere to anywhere. In contrast, in a small world graph, the hubs tend to connect seemingly "distant" vertices in at most two steps.

Of course, path lengths in a small world graph won't be as low as in a complete graph, because if a graph is complete, the user can get anywhere from anywhere in exactly one step: the characteristic path length (the average shortest distance between any two vertices) of a complete graph is exactly 1. At the other extreme, a cycle connects every vertex with the longest path possible: if a graph of N vertices is a cycle, its characteristic path length is $N/2$—on average a user has to go half way around the cycle to find what they are after. In general, the characteristic path length will fall somewhere between 1 and $N/2$, and the smaller it is, the faster the device will be to use, other things being equal. (If a user doesn't know how to find short paths, the fact that there are theoretically short paths may not help. However, the converse is always true: making paths longer makes a device harder to use.)

▷ On calculating the characteristic path length, see section 9.6.2 (p. 302). Other uses of the characteristic path length are discussed in section 10.3.1 (p. 333). The characteristic path length is not the only unexpectedly low measure of a

Box 8.4 Small world friendship graphs Consider a graph of people, where everybody is a vertex, and if two people are friends, then we say there is an edge between the corresponding vertices. This graph, the so-called friendship graph, forms a popular example of small worlds.

The graph isn't exactly random, because people tend to make friends depending on other people's popularity and the friends they know. That is, naturally, we tend to know people who are popular, and we make friends either with them or with people they are friends with. Although the graph is far from complete—nobody can possibly know everybody—it's a small world graph.

The graph is small world because friendly, popular people tend to be people other people become friends with. Popular people become, in small world terms, hubs: bigger and bigger friendship networks grow around them, reinforcing their hubness. If you know a hub, you know at one-step-removed very many friends through that hub. Hubs are worth knowing *because* they are well-connected.

It is said that everybody is separated by at most six people in this world-wide friendship graph. This sounds surprising at first, given how many people there are in the world, and how few of them each of us know. Indeed, it isn't quite true.

The psychologist Stanley Milgram wanted to know how well connected we are, though his definition of connectedness was more like "know of" rather than "friends with." In the 1960s he asked 160 people in Omaha, Nebraska, to write a letter to somebody as close as they knew to a target stockbroker Milgram had chosen in Sharon, Massachusetts. And when that intermediate person got a letter, they were to repeat the process and forward the letter to the next closest person they knew toward the target person.

You would expect the letters to go a long, round-about route—and you'd expect lots of steps to be needed to link Omaha to a nondescript person in Sharon. Milgram found that in fact only about 6 steps were needed, and about half of all letter chains passed through three key people. These key people were hubs of this network.

If it is true that 6 (or thereabouts) is the maximum path length of the friendship graph, then applying our shortest paths algorithm to it would create a matrix of numbers all (roughly) 6 or less.

▷ How to find and calculate shortest paths is discussed in section 9.6 (p. 297).

small world graph. The eccentricities and in particular the diameter are also smaller. These properties are defined in section 9.6.3 (p. 303).

In a complete graph, every vertex has $N - 1$ arcs coming from it, and in a cycle every vertex has one arc coming from it. These are the two extremes. As we add more arcs to an intermediate graph, the characteristic path length will get smaller, but generally we need to add a lot of arcs to get a low characteristic path length. Or we can add arcs preferentially to popular hubs. Then the graph becomes a small world, and the path length will be lower than we'd expect from its number of arcs, at least compared to a random graph. In practical terms, this means that each vertex has a low out-degree (a low number of out-going arcs) yet is closely connected to every vertex.

In interactive device terms, this means that each state needs few buttons (out-going arcs), yet is not too far from any other state. Making a device a small world is a way to make every state easier to get to without greatly increasing the number

Box 8.5 Erdös numbers Another human-grown network is the network of people who write articles together. Paul Erdös was a prolific mathematician who wrote a lot of articles about graph theory, and he coauthored many of his vast output of 1,535 papers.

Anyone who wrote with Erdös is said to have an Erdös number of 1, Erdös himself having an Erdös number of 0. Anyone who wrote an article with someone who has an Erdös number of 1 will have an Erdös number of 2, and so on. I have an Erdös number of 4, because I wrote a paper called "From Logic to Manuals," with Peter Ladkin (it appeared in the *Software Engineering Journal*, volume **11**(6), pp347–354, 1997), who wrote a paper with Roger Maddux, called "On Binary Constraint Problems," in the *Journal of the ACM*, **41**(3), pp435–469, 1994, who wrote a paper with Alfred Tarski (himself a prolific mathematician), who wrote several papers with Erdös, for instance in the *Notices of the American Mathematical Society*. Tarski has an Erdös number of 1; Maddux, 2; Ladkin, 3; and I, 4. That's a low number—but all of us tend to write papers with people who like writing papers with one another. We tend to write papers with authoring hubs, and hubs are well connected and reduce the distances enormously.

Since "standby" is the main state of an interactive device, where every user action must eventually return, perhaps we should be interested in standby numbers—which are a measure of how hard a state is to reach from the standby state.

of buttons. For example, a complete graph of N states needs N buttons, which is impractical for large numbers of states. Moreover, the concept of hubs, which are important states, makes sense for the user. So we win both ways.

We expect to find that interactive devices, especially ones that have been built and revised over a period of years, to be small worlds. We also expect user models to be small worlds too: whatever the graph of the physical device a user is interacting with, they will tend to learn a subgraph of it, but of course as they learn more about the device, they can only add states and arcs to their model when they are actually in those states or traveling those arcs.

A user's mental graph, modeling a system, will grow preferentially as the user learns more, but it grows according to what the user does—users can't know what they don't explore on the device! Whatever states are most popular for the user will be best connected—they will become hubs in the user's model. It's nice that the user's model, being a small world, will give the user efficient ways (short paths) to get from any state to any other state. The user will understand devices faster by building a small world model—and that is exactly what happens!

■ ■ ■

That's a quick overview of small worlds. So what? All small world networks have two very interesting but contrasting properties that are relevant to interaction programming. Small world networks are robust, but they are susceptible to attack.

- In an random, non-small world graph, if we delete a vertex, the average path length will increase. In a small world graph, if we delete a random vertex (or a random edge), average path lengths won't increase much—unless we remove

Box 8.6 LaTeX I used LaTeX to write this book instead of a conventional wordprocessing application like Microsoft Word—LaTeX is a markup language, a little bit like HTML.

If I had used Word, when I wanted a special symbol, say the Greek letter α, I would have to have moved the mouse, clicked on a menu, then muddled around to find the right symbol, then clicked "insert." However, if Word's help system was complete, I could have typed "alpha" into its search box, and found the Greek letter α directly, and perhaps (if I was lucky) a button called [Do it now]—like my dialog box on p. 354.

In fact, LaTeX *already* works like this: I just type `\alpha`, and I get what I want directly. Typing \ is faster than moving the mouse to some "insert symbol" menu—and I can do it by touchtyping without looking at the screen or menus.

In effect, the backslash character \ puts LaTeX into a hub state. Although this makes LaTeX sound better, its disadvantage is that LaTeX itself doesn't provide the user with any help; in Microsoft Word, a user searching for symbols can see menus of the things, whereas in LaTeX users have to know how to spell what they want.

Of course, there is no reason why the best of both worlds could not be combined, but that's a different story . . .

▷ Features like [Do it] are discussed in section 10.7.2 (p. 352).

a hub. But there aren't many hubs, so we are unlikely to pick a hub by chance. Thus if a random vertex "goes down" in a small world network, very little of the connectivity suffers. For example, if we pick a purely random airport, say, Luton, few people (outside of Luton!) would notice if it wasn't working.

- However, if we deliberately kill off a hub, then the average path length will increase a lot. For example, if a terrorist destroys Heathrow, the airport hub, then very many routes would be disrupted. Hopefully, then, we work out where the hubs are and defend them better against failure or against deliberate attack.

For interactive devices, if what the user *doesn't* know is random, then they probably have a good model of the device. However, if the user is missing critical information, namely, information about the hubs, then their knowledge of the device will be very inefficient. So designers should work out where their device's hubs are and make sure that users are aware of them, through the user manual and by using screen design, layout, or other features to make hubs more salient to the user.

Fortunately, there is an easy way to create hubs. An interactive help system is a state cluster that could be connected to every significant state—if every entry in the help system has a [Do it] button. The theory suggests that a user would find the system easier to use because everything becomes easier to find. In fact, it would not be too surprising if users started describing what they wanted in the help system's search dialog box as an easier way than using the normal menu structure.

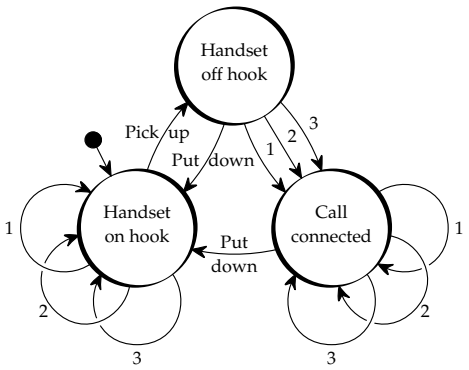


Figure 8.17: This is a UK Coast Guard phone near the sea. Imagine seeing somebody getting into difficulty in the water, so you run to this emergency phone to call the Coast Guard. As you run towards it, the big writing primes you to press 999 (the UK emergency number, as opposed to 911, 112, etc, which are used elsewhere) but when you get closer just *how* are you supposed to do that? It's a serious design issue that could have been found if the user interface had been represented as a graph: there is a transition between two states that has three buttons that are used nowhere else in the design. Giving the user unnecessary choice slows them down, here delaying calls to the Coast Guard—it'd be better to call the buttons 999 anyway! It would be easy to write a program to automatically find such errors, and many others, in the graphs of designs before they are built (as we discuss in following chapters), but unfortunately here it is too late for the the Coast Guard to fix the problem easily.

8.10 Conclusions

Graph theory gives us a very clear way of thinking about interaction programming, and as the bulk of this chapter showed, almost any question about usability design can be rigorously interpreted as a question about graphs. In turn, any rigorous question about graphs can be answered by computer programs analyzing the design.

A graph is a way of representing an interactive device, and once you think of a device as a graph, you can ask all sorts of graph-type questions about its design and figure out how those questions (and their answers) relate to usability and what you want the device to do well.

Few interactive systems are really programmed as explicit graphs, and the programmers end up not having a clue what the interaction graph really is. They then cannot change it reliably, and they certainly cannot verify that it conforms to anything. The result is bad interactive systems, that cannot use computers to help with the design process, for instance, to draft user manuals for technical authors. Almost all graph theory questions are too hard for designers (or even users) to answer without a lot of help from a computer. Since we've shown that lots of graph theory questions are highly relevant to usability, then it follows that designers ought to be using computer-supported design tools that can answer various graph theory questions.

Sure, some graph theory questions take longer to answer, and a few questions take far too long to answer in practice, even on smallish graphs (the traveling salesman is a case in point). The chances are that any question that takes a computer a long time to answer is not going to be very relevant for making the user interface easier to use, since such questions won't have much meaning in the user's head—users aren't going have time to stay around and find out if they take too long to work out. Nevertheless, there's no reason not to use graph theory more.

8.10.1 Further reading

Graph theory is a very rich area, and well worth mining for ideas relevant to interaction programming. I've covered only a very few of the ideas in this chapter. There are many more types of graph (star, de Bruijn, bipartite ...), many more types of graph properties (diameter, radius ...), many more properties of vertices (eccentricity, centers ...) and many more ways of manipulating and combining graphs (cross product, complement, transitive closure ...)—a designer should explore these to see if some ideas match user needs for their devices.

▷ We'll cover a few more graph concepts later in this book as we need them. For example, eccentricity is defined and used in section 9.6.3 (p. 303); we need it because it measures the worst thing a user can try in a given state.

- Bell, T., Witten, I. H., and Fellows, M., *Computer Science Unplugged*, 1999. This book is full of activities that can be done by individuals or groups of people, whether students or school children. This brilliant book is highly motivating and goes against the boring standard ways of teaching graph theory, which all too often make it look as if it is a complex and arcane subject. See unplugged.canterbury.ac.nz to get a downloadable version.

- Biggs, N. L., Lloyd, E. K., and Wilson, R. J., *Graph Theory: 1736–1936*, Cambridge University Press, 1986. As its title suggests, this presents a historical perspective on the development of graph theory, complete with (translated) extracts from key papers. It is an easier read than most books on graph theory because, as it works through history, the earlier stuff is inevitably simpler. theory plunge into definitions and give you everything (I think) too quickly.

- Buckley, F., and Lewinter, M., *A Friendly Introduction to Graph Theory*, Prentice Hall, 2003. This really is a well-written, friendly book on graph theory, but obviously emphasizing the mathematics and going more deeply into graph theory than we can here.
- Michalewicz, Z., and Fogel, D. B., *How to Solve It: Modern Heuristics*, Springer, 2000, is a wide-ranging discussion about problem-solving in programming, with the added benefit that it puts the traveling salesman problem in the context of other problems and techniques.
- Milgram, S., "The Small World Problem," *Psychology Today*, **2**:60–67, 1967. The original paper that established the "six degrees connectivity" of the human race.
- Thimbleby, H. W., "The Directed Chinese Postman Problem," *Software—Practice & Experience*, **33**(11), pp1081–1096, 2003. Unfortunately the Chinese postman tour is a bit too tricky to program to include its source code in this book. This paper gives complete Java code for the Chinese postman tour, as well as further discussion of its history and applications.
- Watts, D., *Six Degrees*, Heinemann, 2003. This is a popular account of small world networks. Note that most discussions of small worlds, including this one, are not very interested in the names of edges—in device terms, the names of buttons. For a user of a device, the names (or appearance) of buttons are very important. For example, what might be a simple cyclic graph (considered as an unlabeled graph) might need different button presses for each edge: the apparent simplicity of the cycle would be lost on the bemused user.

■ ■ ■

Or see the ACM digital library at www.acm.org/dl; Google at www.google.com; or the online encyclopedia, www.wikipedia.com—all hubs of knowledge.